Ruby on Rails

Coloque sua aplicação web nos trilhos

Agradecimentos

Monsters are real, and ghosts are real too. They live inside us, and sometimes, they win.
– Stephen King

Este livro não existiria sem a ajuda dos meus grandes amigos Matheus Bodo, Willian Molinari, Sérgio Schezar e Vinícius Uzêda, que me acompanharam nesse processo quase todos os dias, revisando, criticando e opinando minuciosamente o conteúdo desse livro. Muito obrigado!

Muito obrigado também a família Casa do Código e Caelum, pela oportunidade de escrever esse livro e pelos ensinamentos, especialmente ao Adriano Almeida, pelo difícil trabalho de colocar ordem às minhas palavras.

Agradecimentos especiais também ao GURU-SP (Grupo de Usuários Ruby de São Paulo), à PlataformaTec e aos amigos do ICMC-USP, pois se sei alguma coisa, devo tudo a eles.

Agradeço também a minha família e amigos, pela força e por tolerarem meses sem notícias enquanto me mudo para outro país.

Por fim, agradeço principalmente a você leitor, por investir seu tempo a aprender uma tecnologia que eu pessoalmente gosto tanto. Espero sinceramente que seja uma jornada divertida e lucrativa ao mesmo tempo!

Sumário

Versão: 16.2.23

Capítulo 1

Introdução

Quem, de três milênios, / Não é capaz de se dar conta / Vive na ignorância, na sombra, / À mercê dos dias, do tempo.
– Goethe

Se você desenvolve para a Web, provavelmente já ouviu falar sobre Ruby on Rails. O Ruby on Rails (ou também apenas "Rails") é um *framework open-source* para desenvolvimento de aplicações web, criado por David Heinemeier Hansson (ou "DHH"). O *framework*, escrito na linguagem Ruby, foi extraído de um produto de sua empresa, o Basecamp® (http://basecamp.com) em 2003.

Desde então ele ficou muito famoso, levando também a linguagem Ruby, anteriormente apenas conhecida no Japão e em poucos lugares dos Estados Unidos, ao mundo todo. Mas por que cresceu tanto? O que a linguagem e o *framework* trouxeram de novo para sair do anonimato e praticamente dominar o mercado de *startups* nos Estados Unidos e no mundo?

Quanto ao Rails, na época em que foi lançado, trouxe uma visão diferente ao desenvolvimento Web. Naquele momento, desenvolver para Web era cansativo, os

frameworks eram complicados ou resultavam em sistemas difíceis de se manter e de baixa qualidade.

O DHH, ao desenvolver o Basecamp, pensou principalmente nos seguintes aspectos:

- "Convention over configuration", ou convenção à configuração: ao invés de configurar um conjunto de arquivos XML, por exemplo, adota-se a convenção e apenas muda-se o que for necessário;
- "Dont Repeat Yourself", ou "não se repita": nunca você deve fazer mais de uma vez o que for necessário (como checar uma regra de negócio);
- Automação de tarefas repetidas: nenhum programador deve perder tempo em tarefas repetitivas e sim investir seu tempo em resolver problemas interessantes.

Esses conceitos são amplamente explorados no clássico *The Pragmatic Programmer: From Journeyman to Master [1]*, leitura recomendada. Esta soma de tecnologias e práticas são bastante prazerosas de se trabalhar; é possível realizar muito com pouco tempo e você verá, capítulo a capítulo, como essas ideias são representadas em todos os aspectos do *framework*.

1.1 Nova edição, atualizado para Rails 4!

Na primeira edição, tínhamos duas partes, a primeira, na qual víamos uma introdução a Ruby e, na segunda, cobríamos Rails. Nessa edição, vamos fazer uma coisa diferente. Vou assumir que você, leitor, já sabe Ruby, e vamos cobrir Ruby on Rails 4, recentemente lançado! São muitas novidades, e portanto vamos dedicar o livro completamente ao Rails. Este livro também possui uma revisão completa do texto e do conteúdo técnico, com a colaboração dos leitores da edição anterior.

Se você não sabe Ruby, recomendo que você adquira o livro "Ruby: Aprenda a programar na linguagem mais divertida", de Lucas Souza. Você pode comprar a sua cópia em formato físico ou digital no site da Casa do Código: http://www.casadocodigo.com.br/products/livro-ruby.

1.2 Para quem é este livro

O objetivo deste livro é apresentar um pouco da linguagem Ruby e aprender os primeiros passos a desenvolver com o *framework* Ruby on Rails. Mais além, vamos

aprender a desenvolver aplicativos **com** Rails. Algumas vezes usaremos inclusive componentes feitos em Ruby puro. Esses momentos são muito importantes para aprendermos também algumas boas práticas.

Tendo isso em mente, este livro serve para pessoas que:

- Já leram a primeira edição do livro, mas querem conhecer as novidades da nova versão do Rails;
- Já conhecem as versões anteriores do Rails, mas quer ficar por dentro das mudanças do framework;
- Já conhecem Rails, mas não estão confortáveis em como fazer aplicações bem organizadas;
- Já conhecem Rails superficialmente, mas querem aprimorar conhecimentos e boas práticas.

1.3 Organização

A primeira parte do livro é dedicada a entender o contexto que o Ruby on Rails trabalha. Essa é a parte teórica, na qual vamos entender quais são os principais conceitos por trás do *framework*. Vamos ver também, em alto nível, quais são os componentes do Rails e como eles se relacionam.

A segunda parte é onde vamos fazer a aplicação com Rails. Passo a passo, vamos implementar uma aplicação do início ao fim e, durante a construção de cada funcionalidade, aprender como juntar as partes do *framework*. Vamos comparar o que existia nas versões anteriores e o que veio na nova versão. Vamos revisitar funcionalidades, aprimorando-as, e mostrar como é o processo de criação de uma aplicação real, de forma que você aprenda uma das possíveis maneiras de construir suas aplicações no futuro.

Vamos construir um aplicativo chamado Colcho.net. O Colcho.net é um site para você publicar um espaço sobrando na sua casa para hospedar alguém por uma ou mais noites. O site vai ter:

- Cadastro de usuário, com encriptação de senha;
- Login de usuários;
- Envio de emails;

- Internacionalização;
- Publicação e administração de quartos;
- Avaliação de quartos e *ranking*;
- Busca textual;
- *URL slugs*;
- Uploads e thumbnails de fotos.
- Deploy da aplicação no Heroku

Embora este livro tenha sido construído de forma que a leitura progressiva seja fácil, ele pode servir como consulta. Vamos dividir o sistema que será construído em algumas funcionalidades principais e desenvolvê-las individualmente em cada capítulo.

1.4 Socorro, estou perdido! Me ajude!

Acompanhado desse livro, existe uma lista de discussões no *Google Groups*, localizado em http://colcho.net/lista. Fique à vontade para mandar dúvidas sobre o seu progresso com o livro, ou sobre qualquer coisa relacionada com Rails. Nela, eu e outros leitores voluntários poderão te ajudar com qualquer aspecto relacionado a Ruby, Rails e ao livro. Você pode mandar sugestões, críticas e correções ao livro nessa lista.

Capítulo 2

Conhecendo a aplicação

Para cada fato existe um conjunto infinito de hipóteses. Quanto mais você observa, mais você enxerga.
– Robert Pirsig, em Zen and the Art of Motorcycle Maintenance

Antes de começarmos, você sabe qual a diferença entre um *framework* e uma biblioteca? Quando usamos uma biblioteca, nós programadores escrevemos nosso código, chamando a biblioteca quando necessário. A camada entre a biblioteca e o nosso código é bastante distinta.

O mesmo não acontece quando usamos um *framework*. Para se ter uma ideia, muitas vezes o ponto inicial do sistema não é um código que você escreve. Nosso código faz o intermédio com diversas outras bibliotecas que compõem o *framework*, de forma que o resultado torna-se mais poderoso do que a soma das partes.

O Ruby on Rails não é diferente. Ele é um *framework* usando uma estrutura chamada MVC -- *Model View Controller*, bastante conveniente para a construção de aplicativos Web. O Rails também é usado por muitos desenvolvedores já há bastante tempo, portanto já foi testado em diversas situações, como alta carga, ou grande

número de usuários. É fácil começar com Rails pois ele faz muito trabalho por você, mas se aprofundar no Rails te dá ainda mais poder.

É muita coisa para aprender, mas não se preocupe, este livro está em seu poder justamente para facilitar esta tarefa. Vamos então ver como vamos modelar nossa aplicação e em seguida vamos ver como o Ruby on Rails vai nos ajudar a construí-la.

2.1 Arquitetura de aplicações web

Construir aplicações bem feitas em Ruby on Rails é um pouco mais complicado do que simplesmente criar páginas atrás de páginas. A razão disso é que ele é preparado para criar aplicações modernas e arrojadas. Isso significa que não somente deve responder HTML aos usuários, mas também responder de maneira adequada para aplicações ricas em *client-side*, interagindo com *frameworks* como Backbone.js (http://backbonejs.org/) , Ember.js (http://emberjs.com/) ou outros.

O que são aplicações ricas em client-side?

Depois da repopularização do JavaScript com o uso intensivo de AJAX, os browsers tem se tornado muito mais poderosos do que antigamente, inclusive com os benefícios incorporados pelo HTML 5. Tendo isso em mente, programadores estão criando aplicações cada vez mais complexas, executando grande parte, ou até mesmo totalmente no browser.

Nos últimos anos, diversos *frameworks* tem sido lançados, tal como os mencionados anteriormente. Dessa forma, existem aplicações Web que a parte de *back-end* tornou-se uma camada de persistência que está totalmente independente da apresentação final ao usuário.

2.2 Recursos ao invés de páginas

Para que nossas aplicações fiquem elegantes, precisamos parar de pensar na maneira antiquada de se criar aplicações Web cheias de páginas e interações complexas entre si, e pensarmos em recursos. É uma mudança não trivial, mas vamos usar este pensamento durante o livro todo, então até o final, você ficará mais confortável com essa ideia.

Se você já ouviu falar ou sabe o que é REST (*Representational State Transfer*, ou Transferência de estado de representação), entender como vamos usar recursos para modelar nossa aplicação será bem mais fácil. Se você quiser se aprofundar no assunto, recomendo a leitura do livro *Rest in Practice [4]*.

2.3 Recursos no Colcho.net

Um recurso, em uma aplicação web, é qualquer coisa que uma aplicação pode servir aos seus usuários. Por exemplo, em uma rede social, recursos podem ser mensagens, fotos, vídeos. Uma aplicação web, quando serve um recurso, permite que você interaja com ele. Por exemplo, é possível criar novas mensagens, listá-las ou até mesmo deletá-las. Um recurso também possui uma apresentação, como por exemplo um documento HTML ou um objeto em JSON.

O Colcho.net é um aplicativo em que os seus usuários podem mostrar quartos vagos, ou até um colchonete que pode ser usado em sua sala em troca de uma grana e quem sabe fazer novas amizades.

Assim, o primeiro recurso que podemos identificar é o "Quarto". Em nosso sistema, será possível listar quartos, exibir detalhes de um quarto, criar um novo quarto, editar um quarto já existente e, por fim, removê-lo.

Recursos assim são fáceis de serem mapeados, pois estão diretamente ligados a uma unidade lógica do sistema. Porém, existem recursos que são menos óbvios. Um exemplo de recurso não tão trivial é o recurso "Sessão". Poderemos criar uma sessão ou destruí-la, porém nada mais além disso.

Em sistemas REST, ações com recursos são mapeados em duas partes: um verbo HTTP (uma operação), uma URL (identificação do recurso). Podemos também ter uma representação, porém opcional. Veja os seguintes exemplos:

O que são verbos HTTP?

Verbos HTTP são ações que podem ser executadas em um recurso identificado por uma URL. A implementação do comportamento fica a critério do servidor, não existe um padrão. Os verbos usados no Ruby on Rails são:

- **GET** - para retornar um recurso específico ou uma coleção;
- **POST** - para enviar um novo elemento à uma coleção;
- **PUT** - para alterar um elemento existente;
- **DELETE** - para remover um elemento.

- `GET /rooms.html`: Verbo `GET` na URL `/rooms` mapeia para a listagem de todos os quartos, com uma representação em HTML;
- `POST /rooms`: Verbo `POST` na URL `/rooms` mapeia para a inserção de mais um novo quarto na coleção de quartos, e sua representação é dada pelo cabeçalho HTTP 'Content-type';
- `DELETE /rooms/123`: Verbo `DELETE` na URL `/rooms/123` mapeia para a remoção do quarto cujo identificador único é "123", sendo a representação irrelevante, pois não há a transmissão de dados, apenas uma ação.

O Rails considera esses conceitos em seu design, portanto, se seguirmos esses conceitos, o Rails nos ajuda bastante e de quebra ganhamos facilidade ao usar *frameworks* de *client-side* quando necessário.

Essa é a visão externa da arquitetura de aplicações Web. Vamos entrar então na arquitetura interna.

2.4 Conhecendo os componentes

O Ruby on Rails é um *framework* que adota o padrão de arquitetura chamado `Model-View-Controller (MVC)`, ou Modelo, Apresentação e Controle.

Modelos possuem duas responsabilidades: eles são os dados que sua aplicação usa, normalmente persistidos em um ou mais banco de dados (seu perfil de usuário, por exemplo). Eles também fazem parte da regra de negócio, ou seja, cálculos e outros procedimentos, como verificar se uma senha é válida ou o fechamento de uma fatura.

O **Controle** é a camada intermediária entre a Web e o seu sistema. Ele traduz os dados que vem de representações web (sejam por parâmetros na URL, formulários enviados ou dados JSON enviados por uma aplicação cliente) e repassa para os modelos, que vão fazer o trabalho pesado. Em seguida, preparam os resultados de forma a serem usados na camada de Apresentação.

A **Apresentação** é como o aplicativo mostra o resultado das operações e os dados. Normalmente podem ser uma bela página usando as novas tecnologias de CSS 3 e HTML 5 a até representações de objetos em JSON ou XML.

2.5 Os modelos

Quando criamos aplicações Ruby on Rails, basicamente usamos um componente chamado `ActiveRecord`. O `ActiveRecord` é uma implementação de *Object-Relational-Mapping*, ou seja, uma biblioteca que faz o mapeamento de estruturas relacionais (leia-se SQL e bancos de dados relacionais) a objetos Ruby, transformando, por exemplo, chaves estrangeiras em relacionamento entre objetos através de métodos.

O `ActiveRecord` internamente usa uma biblioteca bastante poderosa chamada `Arel`. Com o uso do `Arel` o `ActiveRecord` consegue transformar chamadas de métodos Ruby em complexas consultas SQL e depois mapear de volta em objetos Ruby. Isso tudo é muito conveniente, veja alguns exemplos:

```
Room.all
#  SELECT "rooms".* FROM "rooms"

Room.where(:location => ['São Paulo', 'Rio de Janeiro'])
# SELECT "rooms".* FROM "rooms" WHERE "rooms"."location"
#                                IN ('São Paulo', 'Rio de Janeiro')
```

Além disso, o `ActiveRecord` traz diversas facilidades para integração com o restante de uma aplicação web, através do `ActiveModel`. O `ActiveModel` é um componente que inclui diversas ferramentas para validação de atributos, *callbacks*

em momentos oportunos (como antes de atualização ou criação), integração com formulários, tradução de atributos, entre outras.

Os modelos não necessariamente precisam ser objetos `ActiveRecord`. É bastante comum separarmos regras complexas em diversas classes em Ruby puro. Isso é importante para evitar que os modelos ou controles fiquem grandes e complexos. Vamos ver um exemplo disso no capítulo "9 Login do usuário".

No caso do Colcho.net, o Quarto se encaixa perfeitamente como um modelo. Assim, podemos criar validações como a presença de uma localidade ou calcular a disponibilidade dele em uma certa data.

2.6 Controle

A camada de controle é o intermédio entre os dados que vem dos usuários do site e os Modelos. Outro principal papel da camada de controle é gerenciar sessão e cookies de usuário, de forma que um usuário não precise enviar suas credenciais a todo momento que fizer uma requisição.

Após obter os dados na camada de modelos, é papel da camada de Controle determinar a melhor maneira de representar os dados, seja via a renderização de uma página HTML ou na composição de um objeto JSON.

No Ruby on Rails, os componentes que trabalham nessa camada são o `ActionDispatch` e o `ActionController`, ambos parte do pacote denominado `ActionPack`. O `ActionDispatch` trabalha no nível do protocolo HTTP, fazendo o *parsing* dos cabeçalhos, determinando tipos MIME, sessão, cookies e outras atividades.

Já o `ActionController` dá ao desenvolvedor o suporte para escrever o seu código de tratamento das requisições, invocando os modelos e as regras de negócio adequadas. Ele também dá suporte a filtros, para, por exemplo, verificar se um usuário está logado no sistema para fazer uma reserva no site.

O `ActionDispatch` é um componente de menor nível, então não vamos interagir diretamente com este componente. Porém, o `ActionController` é um dos componentes centrais de aplicações Rails, então vamos, a todo momento, usar facilidades dele.

2.7 Apresentação

A camada de apresentação é onde prepara-se a resposta para o usuário, depois da execução das regras de negócio, consultas no banco de dados ou qualquer outra ta-

refa que sua aplicação deva realizar.

As maneiras mais comuns de se exibir dados atualmente na web é através de páginas HTML. No Ruby on Rails, conseguimos construí-las com o auxílio da biblioteca `ActionView` (também membro do `ActionPack`). Também é possível ter a representação JSON dos dados, via a serialização de objetos (transformação de objetos Ruby puro em JSON) ou via a construção de templates com o `JBuilder`, por exemplo.

Depois que a apresentação final é montada, ela é entregue ao usuário via um servidor web, como por exemplo, uma página mostrando todas as informações de um quarto, ou um objeto JSON.

Por fim, o Rails ainda possui uma estrutura complexa para gerenciar imagens, *stylesheets* e javascripts, chamada `Asset Pipeline`, pré-processando os arquivos e preparando-os para entregar da melhor forma para o usuário.

2.8 Rotas

É necessário informar ao Rails quais as URLs que a aplicação é capaz de responder e qual Controle deverá ser invocado. Esses mapeamentos são chamados de rotas. O mapeamento de rotas no Rails é baseado na forma de recursos (veja na seção 2.3), mas é possível também fazer rotas customizadas.

Veja a figura 2.1. Nela é possível ver a interação entre as diversas camadas MVC e o roteador quando o usuário requisita uma página de edição do recurso “quarto":

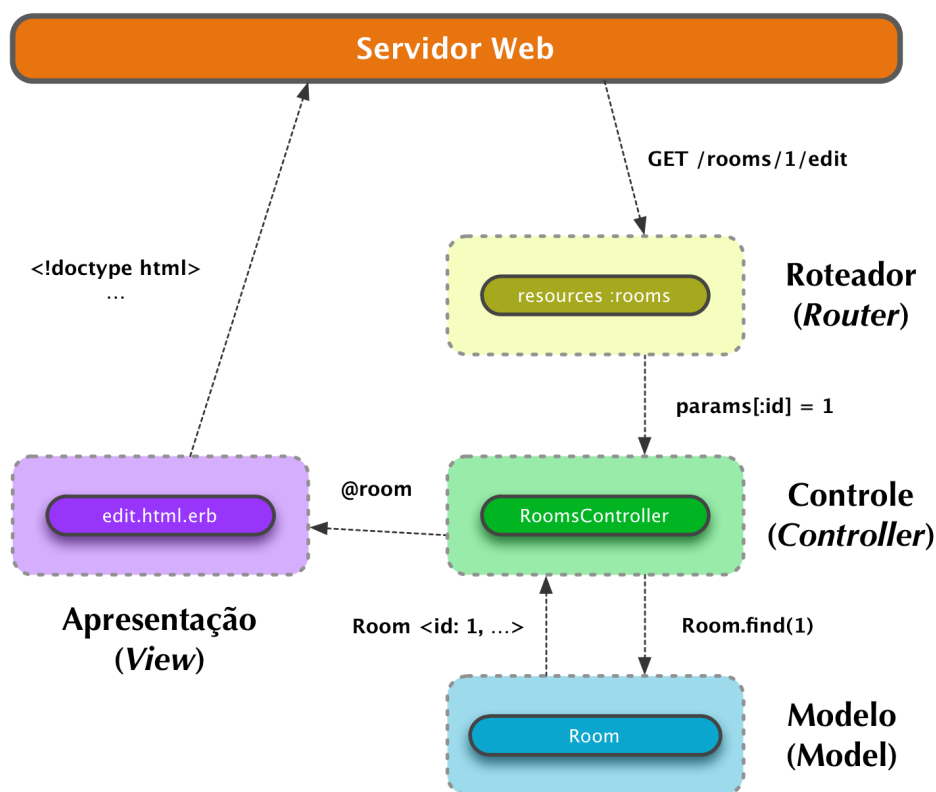



Figura 2.1: Fluxo do MVC

2.9 Suporte

Em volta de todo os componentes que mencionamos antes, temos o `ActionSupport`. O `ActiveSupport` não é uma camada em si, pois está presente em todas as outras camadas. É no `ActiveSupport` que ficam diversas ferramentas e extensões ao Ruby, para deixá-lo ainda mais fácil de usar. Veja o exemplo a seguir:

```
2.days.ago
# => Sun, 03 Jun 2012 01:08:33 BRT -03:00
```

Essa linha de código imprime a data de dois dias atrás. Essa funcionalidade não existe por padrão no Ruby, mas, por ser bastante conveniente, o `ActiveSupport`

faz essa extensão para você.

Existem vários outros métodos implementados pelo `ActionSupport`, mas não vale a pena entrar em detalhe sobre cada método. Vale a pena ler a documentação oficial sobre esse módulo em: http://guides.rubyonrails.org/active_support_core_extensions.html.

2.10 Considerações finais

Não se preocupe em decorar todos os nomes que foram vistos nesse capítulo. Você pode consultar o livro sempre que precisar saber. É importante porém saber que eles existem para que você possa procurar ajuda ou documentação.

Aliás, falando em documentação, você aprenderá com o decorrer do livro que a API do Rails é bem extensa e cheia de métodos interessantes. Ambiente-se a desenvolver com o site da documentação oficial do Rails aberta. Ela está disponível em http://api.rubyonrails.org/. Sempre que tiver alguma dúvida sobre algum método do Rails, você pode procurar que estará lá. Outra ferramenta bastante útil são os Rails Guides, ou o Guia do Rails, ótimo para tirar dúvidas gerais sobre cada componente. O Guia do Rails fica em http://guides.rubyonrails.org/.

O que é extremamente importante entender desse capítulo, porém, são as ideias de recursos e entender o que cada parte do MVC faz devidamente. Isso significa que, sempre que possível, devemos modelar nossas aplicações como recursos e que **nunca**, **jamais**, devemos colocar regras importantes de negócio em Controles, por exemplo, pois não é o papel dele.

Entendido isso, vamos continuar. Agora é hora de voltar a programar!

Capítulo 3

Primeiros passos com Rails

Conhecer seus limites já é estar a frente deles.
– Georg Wilhelm Friedrich Hegel

3.1 Instalação do Rails

Para começar, é necessário ter o Ruby e o Ruby on Rails para continuar. Veja a seguir instruções específicas para cada Sistema Operacional.

Instalação no OSX

Para quem está iniciando com Ruby on Rails, a forma mais simples é usar o RVM, ou Ruby enVironment Manager. Antes de instalar o RVM, porém, é necessário instalar os pacotes de desenvolvimento da Apple.

Se você não tem o Xcode, vá na página do Apple Developer Tools (https://developer.apple.com/downloads/index.action) e baixe o 'Command Line Tools for Xcode', versão mais atual e siga os passos de instalação. Se tiver o Xcode mais re-

cente, você pode abri-lo, ir nas Preferências > Downloads e instalar o 'Command Line Tools'.

Depois de instalar o Command Line Tools, basta executar o seguinte comando no terminal:

```
curl -L get.rvm.io | bash -s stable --rails
source $HOME/.rvm/scripts/rvm
```

Aproveite e vá tomar um café, vai demorar um pouco. Ao finalizar, este comando deixará instalado tudo que você precisa para começar a desenvolver com Ruby on Rails.

Instalação no Linux

Tanto quanto usuários OSX, a forma mais simples de começar com Ruby on Rails no Linux é usar o RVM, ou Ruby enVironment Manager.

Primeiramente é necessário instalar os pacotes de desenvolvimento do seu sistema. Procure o manual da sua distribuição para saber como instalar. No Ubuntu, por exemplo, basta instalar o pacote `build-essential` e mais algumas outras bibliotecas de dependências:

```
sudo apt-get install build-essential libreadline-dev libssl-dev curl \
libsqlite3-dev
```

Em seguida, basta executar:

```
curl -L get.rvm.io | bash -s stable --rails
source $HOME/.rvm/scripts/rvm
```

Este comando irá instalar o RVM e também, automaticamente, irá instalar a versão mais atual do Ruby e do Rails e todas as dependências. Aproveite para tomar um café ou um chá, pois demora um pouco.

Instalação no Windows

Infelizmente não há maneira fácil de se instalar o Ruby on Rails 4 no Windows. O instalador RailsInstaller (http://www.railsinstaller.org) ainda não está atualizado e portanto, a recomendação é usar uma distribuição Linux virtualizada. :(

3.2 Gerar o alicerce da aplicação

No final deste capítulo, vamos ter o esqueleto preparado para começar o Colcho.net. Para criar um novo projeto, precisamos executar no terminal o comando `rails new`, passando em seguida o nome do projeto que será criado, no caso, `colchonet`.

Ao começar a instalar, aproveite para pegar mais café ou sua bebida favorita, pois pode demorar um pouco.

```
$ rails new colchonet
     create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      ...
      Your bundle is complete! Use 'bundle show [gemname]'
               to see where a bundled gem is installed.
```

Como vimos no Capítulo 2, vamos escrever código dentro do framework. Por causa disso, o framework impõe uma estrutura de pastas (mais uma aplicação da ideia Convenção sobre configuração) que você deverá seguir. Algumas vezes, inclusive, você deverá inserir código dentro de arquivos gerados pelo framework, outras vezes iremos criar novos arquivos.

A estrutura de pasta do Rails é bastante importante, e cada pasta possui uma utilidade específica. Veja a estrutura de pastas gerada nas figuras 3.1 e 3.2.

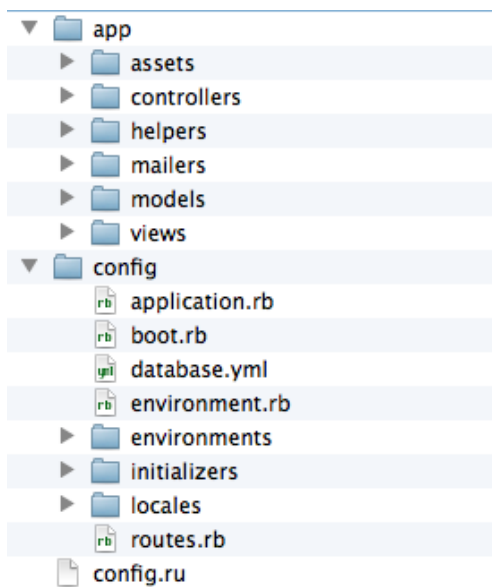



Figura 3.1: Estrutura de pastas geradas - 1

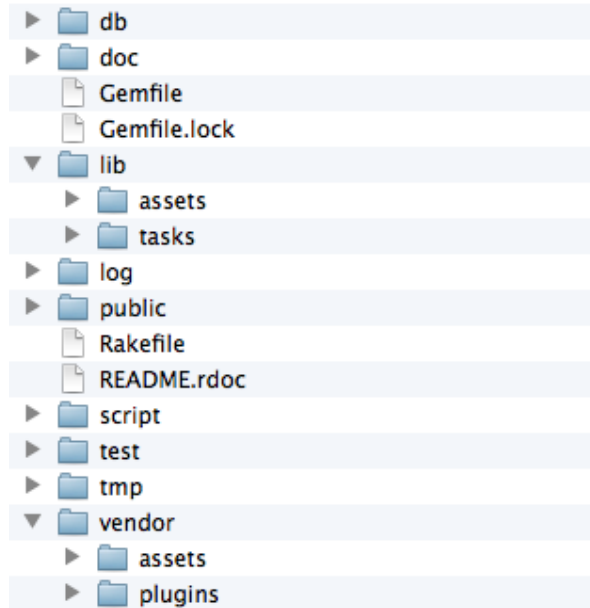



Figura 3.2: Estrutura de pastas geradas - 2

Vamos entender o que é cada uma delas:

- **app** - É onde fica o código principal da sua aplicação. Controles ficam na pasta *controllers*, apresentações em *views*, javascript e CSS em *assets* e modelos em *models*. A pasta *helpers* guarda códigos auxiliares usados em apresentações e *mailers* são classes para o envio de emails.

- **config** - Configuração da aplicação, como informações sobre banco(s) de dados, internacionalização de strings (I18n), time zone, e outras coisas.

- **db** - Armazena migrações e o esquema do banco de dados usado na aplicação.

- **doc** - Toda a documentação de sua aplicação deve estar aí.

- **lib** - Armazena código externo à sua aplicação mas que não faz parte de gems. Aí também ficam tarefas `rake` customizadas, código que deve ser executado fora do ambiente Web.

- **log** - Onde ficarão seus logs e, se tudo der certo, uma pasta que você raramente terá que acessar.

- **public** - Arquivos estáticos como `favicon.ico`, `robots.txt`, `404.html` e `500.html`, que serão exibidos nos casos de página não encontrada e erro de servidor, respectivamente.
- **script** - Guarda o script `rails`, usado para gerar código (vamos ver mais detalhes sobre esse comando a seguir).
- **test** - Testes unitários, integração, funcionais e performance ficam aqui. Este é um tópico bastante complicado e merece seu próprio livro, então, ao invés de vermos o assunto pela metade, indico a leitura do *RSpec Book [3]* ou o *Guia Rápido de RSpec [5]*.
- **tmp** - Nesta pasta ficam arquivos temporários como PIDs, cache, entre outros.
- **vendor** - É onde você deverá instalar plugins de terceiros que não são gems. Também é uma boa ideia instalar eventuais plugins javascript e CSS aqui, para separar o código da sua aplicação.

Pasta vendor/plugins

Nas versões anteriores ao Rails 4, era possível instalar *plugins* na pasta `vendor/plugins`, porém essa funcionalidade foi removida. Atualmente, os *plugins* devem existir na forma de *gems*.

Dê uma olhada no conteúdo das pastas e abra alguns arquivos, apenas por curiosidade, para quebrar o gelo, já que você e o Rails ainda são estranhos entre si. Dê uma atenção especial ao arquivo `config/application.rb`, o arquivo de configurações gerais da sua aplicação.

3.3 Os ambientes de execução

O Rails possui o conceito de ambientes de execução do aplicativo e cada um deles possui um conjunto próprio de configurações. Os ambientes existentes no Rails são:

- `development`: É o ambiente padrão, onde o Rails é totalmente configurado para facilitar o desenvolvimento, localmente em nossos computadores;
- **test** - O ambiente `test` é o ambiente carregado quando são executados os testes unitários e integração;

- **production** - O ambiente `production` (*produção*) é o ambiente em que a aplicação deverá ser executada quando exposta ao mundo e os dados devem ser reais.

Um exemplo de configuração que varia por ambiente é o *caching* de classes, ou seja, todo o código Ruby presente na pasta `app` será carregado apenas uma vez, caso o *caching* esteja ligado. Desligar essa configuração é bastante útil em desenvolvimento pois, uma vez que salvamos um arquivo Ruby modificado, a próxima requisição Web já irá executar o novo código. Esse processo é bastante lento e portanto, devemos desligá-lo no ambiente de produção, senão vamos tornar a aplicação muito mais lenta.

Cada ambiente deverá possuir seu próprio banco de dados. O arquivo `config/database.yml` possui a definição da configuração de banco de dados para cada ambiente. Veja o exemplo para o ambiente de desenvolvimento, usando o adaptador para o PostgreSQL:

```
development:
  adapter: postgresql
  database: colchonet_dev
  host: localhost
  username: vinibaggio
  password:
  pool: 5
  timeout: 5000
```

Essa separação é bastante importante pois não queremos que dados usados em desenvolvimento interfiram com dados em teste ou em produção, ou que, quando executamos os testes unitários, o banco de desenvolvimento seja apagado.

Outros arquivos relacionados com ambientes ficam na pasta `config/environments`, no qual cada arquivo é carregado automaticamente de acordo com o ambiente em execução.

Variável de ambiente RAILS_ENV

A variável de ambiente `RAILS_ENV` é usada para determinar em que ambiente o Rails irá executar, por padrão sendo `development`. Assim, é possível executar comandos específicos para cada ambiente, basta fazer, por exemplo:

```
RAILS_ENV=production rake db:create
```

Dentro da aplicação, é possível saber qual ambiente estamos executando através do método `Rails.env`:

```
Rails.env
# => "development"

Rails.env.production?
# => false

Rails.env.development?
# => true
```

3.4 Os primeiros comandos

Com seu terminal aberto na pasta do projeto (`colchonet`), execute o comando `rails server`:

Comandos no terminal

A partir de agora vamos ver diversos comandos de terminal e suas respostas. Por isso, nessas listagens você deve digitar apenas o que segue o símbolo $ (dinheiro), ou se por um acaso a linha for muito longa, será quebrada com o símbolo \(contra-barra). Exemplos:

```
$ rails server
=> Booting WEBrick
```

Nesse caso, você deverá digitar apenas `rails server` e apertar a tecla Enter. O que não tiver esse prefixo será uma saída aproximada do que você vai ver ao digitar o comando.

```
$ rails server
=> Booting WEBrick
=> Rails 4.0.0 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
[2013-07-16 09:39:09] INFO  WEBrick 1.3.1
[2013-07-16 09:39:09] INFO  ruby 1.9.3 (2012-11-10) [x86_64-darwin12.2.0]
[2013-07-16 09:39:09] INFO  WEBrick::HTTPServer#start: pid=6874 port=3000
```

Abra o seu browser em http://localhost:3000/ e *voilà*, você verá uma página preparada pela equipe do Ruby on Rails te felicitando por estar rodando Rails. Nessa página, há algumas sugestões de passos, como usar o comando `rails generate` para criar modelos e controles.

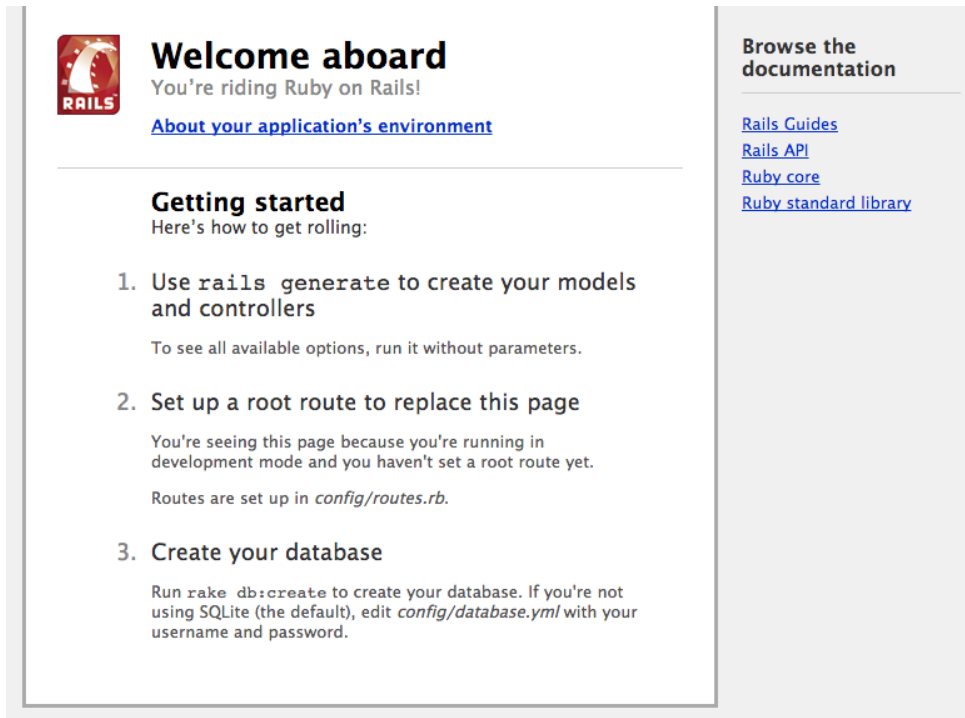



Figura 3.3: Seja bem vindo ao Ruby on Rails

O comando `rails generate` é um comando bastante útil. Se você digitá-lo sem parâmetros, pode ver uma lista do que pode ser gerado. Essa lista pode aumentar, dependendo das `gems` que você instalar no seu projeto (bastando adicionar no arquivo `Gemfile`).

O Rails inclui várias ferramentas para você e o *script* de geração de templates é um exemplo de como a ferramenta te ajuda a automatizar tarefas repetitivas. Nesse espírito, vamos agora começar a criar nosso primeiro recurso no sistema, o recurso quarto, ou `Room`, que terá um título, uma descrição e uma localização. Faremos isso através do comando `rails generate scaffold`, que irá gerar vários arquivos:

```
$ rails generate scaffold room title location description:text
invoke  active_record
  create    db/migrate/20130716164333_create_rooms.rb
  create    app/models/room.rb
  invoke    test_unit
  create      test/models/room_test.rb
```

```
create      test/fixtures/rooms.yml
invoke  resource_route
 route    resources :rooms
invoke  scaffold_controller
create    app/controllers/rooms_controller.rb
invoke    erb
create      app/views/rooms
create      app/views/rooms/index.html.erb
create      app/views/rooms/edit.html.erb
create      app/views/rooms/show.html.erb
create      app/views/rooms/new.html.erb
create      app/views/rooms/_form.html.erb
invoke    test_unit
create      test/controllers/rooms_controller_test.rb
invoke    helper
create      app/helpers/rooms_helper.rb
invoke      test_unit
create        test/helpers/rooms_helper_test.rb
invoke    jbuilder
create      app/views/rooms/index.json.jbuilder
create      app/views/rooms/show.json.jbuilder
invoke  assets
invoke    coffee
create      app/assets/javascripts/rooms.js.coffee
invoke    scss
create      app/assets/stylesheets/rooms.css.scss
invoke  scss
create    app/assets/stylesheets/scaffolds.css.scss
```

Com um simples comando, o Rails gerou muitos arquivos. Antes de entrar em detalhe o que é cada um deles, vamos primeiro ver detalhadamente o comando que executamos, mostrado na figura 3.4:

Figura 3.4: Parâmetros do comando rails

Nessa figura vemos a composição do comando anterior. Fazemos o `generate` de um *scaffold*, ou "andaime". *Scaffolds* são interessantes pois já geram desde o modelo até a apresentação e páginas HTML, para que possamos ter uma base simples e funcional. Isso é bastante conveniente, pois alterar uma base pronta para deixar da forma que quisermos é muito mais simples do que começar do zero, principalmente quando estamos conhecendo o framework.

A próxima parte do comando é onde especificamos o nome do recurso e seus atributos. Por padrão, todos os campos serão "string", ou um campo de texto curto. Porém, ainda é possível especificar o tipo, como é o caso de `description`, a descrição de um quarto pode ser bem longa. Para isso, podemos usar o `:` para especificar o tipo daquele dado.

O comando *scaffold* já gera tudo que é necessário para que possamos fazer as operações mais comuns em um recurso, o famoso CRUD. CRUD nada mais é do que *create*, *retrieve*, *update* e *delete*, ou criar, buscar/listar, editar e remover. Para tal, primeiro precisamos criar o banco de dados. Não se preocupe em configurar um servidor MySQL ou PostgreSQL por enquanto, pois vamos usar um banco de dados bom e simples para o nosso propósito no momento, o SQLite. Para criar o banco de dados e deixar tudo pronto para ver o resultado do *scaffold*, basta executar, na pasta do projeto, os comandos `rake db:create` e `rake db:migrate`:

```
$ rake db:create

$ rake db:migrate
==  CreateRooms: migrating ======================
-- create_table(:rooms)
   -> 0.0017s
==  CreateRooms: migrated (0.0022s) =============
```

Combinando tarefas rake

A aplicação Rails toma um tempo para iniciar e isso pode se tornar um pouco inconveniente conforme a aplicação cresce. Dessa forma, é possível combinar os comandos em apenas uma chamada. Isso irá fazer com que as tarefas rake compartilhem alguns dos passos que são repetidos quando executamos em duas linhas separadas:

```
$ rake db:create db:migrate
==  CreateRooms: migrating ========================
-- create_table(:rooms)
   -> 0.0017s
==  CreateRooms: migrated (0.0022s) ===============
```

`rake db:create` irá criar o banco de dados, se não existir. No caso do SQLite, o comando irá criar os arquivos `db/development.sqlite3`, `db/test.sqlite3` e `db/production.sqlite3`.

Em seguida, o `rake db:migrate` irá criar a tabela `rooms` no banco de dados usando a migração gerada. Veremos no futuro o que é uma migração, não se preocupe se você não souber o que é isso.

Comando rake e comando rails

O `rake`, ou "ruby make", é uma ferramenta para executar tarefas de manutenção de uma biblioteca ou aplicação, independente do Rails. O Rails, porém, adiciona diversas tarefas pertinentes, como criação e migração de banco de dados, executar testes unitários, entre outros. Para saber todos os comandos `rake` que você pode executar no seu aplicativo, execute `rake -T`.

O comando `rails`, porém, em geral executa *scripts* que não dependem do ambiente da sua aplicação, ou seja, não dependem de conexão com banco de dados ou carregar o seu código Ruby, com exceção do `server`.

Agora, vamos executar o servidor:

```
rails server
```

Abra um browser em http://localhost:3000/rooms e você será apresentado pela tela padrão gerada pelo *scaffold*:

Listing rooms

Title Location Description

New Room

Figura 3.5: Página de quartos

Aproveite para dar uma navegada, clique nos links e preencha os formulários. Funcional, apesar de simples. Agora vamos passo a passo inspecionar os arquivos gerados pelo gerador.

Logs

Toda a requisição que você fizer, você vai ver que o servidor do Rails mostra um **monte** de coisas, desde consultas SQL executadas até quando o servidor serve arquivos estáticos. Todo este texto será enviado também para a pasta `log`, no arquivo com o nome do ambiente em execução (por exemplo, `log/development.log`):

```
Started GET "/rooms" for 127.0.0.1 at 2012-08-14 21:58:15 -0700
Processing by RoomsController#index as HTML
  Room Load (0.2ms)  SELECT "rooms".* FROM "rooms"
  Rendered rooms/index.html.erb within layouts/application (2.1ms)
Completed 200 OK in 12ms (Views: 10.3ms | ActiveRecord: 0.2ms)
```

É sempre importante estar de olho nessas requisições. Podemos observar os parâmetros enviados pelo usuário, as consultas SQLs resultantes, o tempo gasto em cada parte da aplicação e o código de resposta enviado ao usuário.

3.5 Os arquivos gerados pelo scaffold

Vimos há pouco tempo que o `scaffold` gerou vários arquivos para nós. Vejamos agora o que significa cada um deles, o que fazem e qual o seu impacto no projeto.

Migrações

Migrações são pequenos *deltas* de um banco de dados, ou seja, elas registram as modificações que o *schema*, ou esquema, do banco de dados vai passando. Uma migração contém três informações importantes: uma informação de versão, um código de regressão e um código de incremento e todas elas ficam dentro do diretório `db/migrate` do projeto.

O código de incremento de versão fica em um método chamado `up`. Quando houver a necessidade de se regredir uma versão, o método executado será o `down`. O nome da versão da migração encontra-se no nome do arquivo, que nada mais é do que uma data (ano, mês, dia, hora, minuto e segundo).

O Rails é capaz de saber o comando de regressão para um comando de incremento. Por exemplo, o comando `drop_table` é o oposto do comando `create_table`. Assim, é possível usarmos o método `change` em uma migração. Nesse método, escrevemos o comando de incremento e, quando precisarmos fazer uma regressão, o Rails automaticamente sabe o comando análogo a ser executado.

Para desfazer uma migração, você pode executar `rake db:rollback`:

```
$ rake db:rollback
==  CreateRooms: reverting ====================
-- drop_table("rooms")
   -> 0.0010s
==  CreateRooms: reverted (0.0012s) ===========

$ rake db:migrate
==  CreateRooms: migrating ====================
-- create_table(:rooms)
   -> 0.0074s
==  CreateRooms: migrated (0.0076s) ===========
```

Veja a pasta `db/migrate`. Lá haverá um arquivo que inicia com uma data, que depende da hora que você executou o comando, e termina com `_create_rooms.rb`:

```
class CreateRooms < ActiveRecord::Migration
  def change
```

```
    create_table :rooms do |t|
      t.string :title
      t.string :location
      t.text :description

      t.timestamps
    end
  end
end
```

Vimos que, para executar as migrações pendentes, basta executar:

```
rake db:migrate
```

No capítulo 4, vamos revisitar e aprofundar em outros comandos de migrações.

Modelo

O arquivo `room.rb` gerado pelo *scaffold* no diretório `app/models` é o modelo que tem seus dados guardados no banco de dados quando você clica no botão "Create Room". Nesse arquivo também ficam as regras de validação, relacionamentos com outros modelos e outras coisas. O código atual é simples e veremos como customizá-lo em breve.

```
class Room < ActiveRecord::Base
end
```

Os próximos dois arquivos são relacionados a testes.

Teste unitário

Testes unitários são artefatos muito importantes para um sistema de qualidade. Escrever teste, porém, é algo bastante complicado para iniciantes, pois existe muita subjetividade em como escrever um teste. Por exemplo, devemos escrever um teste antes do código de produção (ou seja, código que vai ser de fato executado quando o site estiver no ar) ou depois? Devo usar qual técnica? Como usar `mocks` e quando usar `stubs`?

Esse assunto deve ser abordado por si só, dada a sua importância, portanto recomendo a leitura dos livros *The RSpec Book, [3]* e o *Test-Driven Development: By Example [2]*. A comunidade Rails leva testes tão a sério que o próprio *framework* vem com um conjunto de ferramentas para auxiliar o desenvolvedor a criar criá-los.

Um exemplo disso são os arquivos gerados. Toda vez que o Rails gera um modelo, controle ou apresentação, arquivos equivalentes para cada unidade são criados. Outro artefato gerado são as *fixtures*, ou acessório, em uma tradução grosseira. As *fixtures* são arquivos no formato YAML (http://yaml.org/) , uma linguagem simples para serialização de dados, que será inserido no banco de dados antes de executar os testes unitários. Eles são úteis para testar consultas ou para construir um cenário em que um código deve ser executado para que tudo funcione (por exemplo, para testar uma autenticação, é necessário que haja um usuário no banco de dados).

Rotas

Prosseguindo, a próxima modificação é no arquivo de rotas. O arquivo de rotas é onde o Rails mapeia a URL da requisição à um controle que você escreve. Se você se lembra bem, falamos na seção 2.2 que o Rails é bastante voltado à recursos. Isso pode ser observado no arquivo `routes.rb`, no diretório `config`:

```
resources :rooms
```

Essa linha é tudo o que você precisa dizer para ganhar as seguintes rotas:

- `GET /rooms` - ação **index** - lista todos os quartos disponíveis;
- `GET /rooms/new` - ação **new** - mostra uma página com um formulário para a criação de novos quartos;
- `POST /rooms` - ação **create** - cria um novo recurso na coleção de quartos;
- `GET /rooms/:id` - ação **show** - exibe detalhes de um quarto cuja chave primária (famoso id) seja especificada na URL (id 123 para a URL /rooms/123, por exemplo)
- `GET /rooms/:id/edit` - ação **edit** - exibe o formulário para a edição de um quarto;
- `PATCH /rooms/:id` - ação **update** - altera alguma informação do recurso cujo id seja especificado pelo parâmetro :id;
- `PUT /rooms/:id` - ação **update** - o mesmo que o `PATCH`
- `DELETE /rooms/:id` - ação **destroy** - destrói o objeto identificado pelo parâmetro :id

Essas ações são geradas por padrão e são todas as ações mais básicas que queiramos fazer em um recurso (o "CRUD"). É claro que dá para customizar, remover algumas dessas ações ou fazer URLs mais interessantes, mas sempre que fizermos recursos dessa forma, vamos poupar trabalho.

Por que PUT e PATCH?

O Rails 4 introduz o suporte ao método `PATCH` em atualizações de recursos, sendo que, nas versões anteriores, já existia o suporta ao método `PUT` para o mesmo fim.

A razão disso é que o método `PATCH` é uma adição mais recente ao protocolo HTTP e tem um significado um pouco diferente do `PUT`. A ideia é que, quando executamos um `PUT`, o recurso identificado pela URL será completamente substituído pelo objeto sendo entregue, enquanto o `PATCH` suporta atualizações parciais, ou seja, o que será alterado é apenas o que foi enviado.

A diferença pode parecer um pouco sutil, mas pode ser vista da seguinte maneira: o resultado final de um recurso sempre será o mesmo após o resultado de um método `PUT`, enquanto no `PATCH`, o resultado nunca será previsível, pois só temos o controle de uma parcela do conteúdo a ser atualizado.

Por fim, este comportamento de `PATCH` e `PUT` é uma especificação de serviços REST que, por fim, cabe a nós desenvolvedores implementar o comportamento apropriado. O Rails, por padrão, não diferencia os dois métodos e acaba sempre atualizando os recursos parcialmente.

Controle

O próximo conjunto de modificações pertence ao controle em si. No código do controle, você pode observar exatamente as sete ações mencionadas anteriormente e um código que realiza as operações da maneira mais simples possível.

Apresentações

Em seguida, temos as seguintes páginas, apresentações de *algumas* das ações mencionadas:

```
create      app/views/rooms/index.html.erb
create      app/views/rooms/edit.html.erb
create      app/views/rooms/show.html.erb
```

```
create      app/views/rooms/new.html.erb
create      app/views/rooms/_form.html.erb
```

Há três coisas importantes nessa lista. A primeira é que você pode ver que que os arquivos são todos terminados em "erb". O ERB é a linguagem de *template* padrão do Rails, ou seja, é possível embutir código Ruby em páginas HTML para que possamos gerar páginas dinamicamente, dependendo deste código. Além do ERB, o Rails pode usar outras linguagens de *template*, como HAML e Liquid, bastando que você instale os plugins apropriados.

Para simplificar, vamos usar ERB durante esse livro, então não se preocupe com outra linguagem de template por enquanto. Ela não é nada de especial na verdade, é o Ruby comum embutido dentro de tags HTML.

A segunda observação é que não há sete ações e que não necessariamente todas as páginas mapeiam para uma ação no controle/rota. Isso deve-se ao fato que algumas rotas não possuírem uma apresentação em si, apenas redirecionam ou apresentam outras. Um exemplo é a ação **create**, que, quando há sucesso, redireciona o usuário à ação **index**, e quando há erro, o formulário da ação **new** é reapresentado, com a adição de marcação de erro.

A terceira e última observação é que há um arquivo iniciado com _. Isso significa que essa página é uma *partial*, ou seja, o conteúdo dela é embutido em outras páginas. Nesse caso, o `_form.html.erb` é incluído tanto na página `new.html.erb` quanto em `edit.html.erb` pois o formulário é o mesmo, evitando problemas de `Ctrl-C Ctrl-V` e duplicidade de código. Alterando em um lugar, as mudanças se refletem em todas as páginas que usam essa *partial*.

Agora o projeto está criando vida. Vamos então à nossa primeira funcionalidade: cadastro de usuários.

Capítulo 4

Implementação do modelo para o cadastro de usuários

É bom perseguir um objetivo para uma jornada, mas no fim o que realmente importa é a jornada em si.
– Ursula Guin

Nossa aplicação está começando a tomar alguma forma. Já temos um recurso, quartos, e já podemos cadastrá-lo, editá-lo e removê-lo e fizemos tudo isso com a ajuda de geradores de código.

Mas agora que já temos uma breve noção da estrutura do nosso projeto e também do próprio *framework*, podemos começar a construir uma funcionalidade do início ao fim, e bastante importante para o Colcho.net: cadastro e autenticação de usuários, o login.

Primeiramente, vamos construir o modelo de usuários. Vamos garantir integridade das informações, ou seja, construir validações para que um usuário cadastrado com sucesso no site possua todos os dados necessárias.

Uma vez que tivermos o modelo pronto, vamos montar as páginas de cadastro e edição de perfil e por fim, visualização de perfil. Durante esse trabalho, vamos montar as rotas e construir as ações do controle sem o auxílio de geradores.

Em seguida, vamos aprimorar o modelo, garantindo que o usuário possua uma senha encriptada no banco de dados, para que ninguém, nem mesmo os administradores do sistema, consigam acesso a uma informação tão sigilosa.

Os templates estarão bem crus, então vamos aplicar um pouco de CSS e HTML para tornar as telas mais interessantes. Vamos aprender como fazer tudo isso, e, em seguida, iremos aprender a usar o sistema de internacionalização do Rails para traduzir as páginas para múltiplos idiomas.

Para concluir o cadastro de usuários, vamos enviar um e-mail para o usuário para que ele possa confirmar sua conta, uma prática muito comum nos sites atuais para evitar cadastros falsos.

4.1 O usuário

Para o modelo de usuários, vamos precisar dos seguintes campos:

- Nome completo
- E-mail
- Senha
- Localidade
- Bio

Para criar esse modelo, vamos usar o gerador de modelos. Ele é bastante útil, pois já cria a classe `User` e também já gera a migração:

```
$ rails generate model user full_name email password location bio:text
```

Geradores

Você pode observar que muitas vezes vamos usar os geradores que o Rails provê. Porém, é completamente possível desenvolver sem o uso de nenhum gerador e criar todos os arquivos necessários na mão. Recomendo tomar esse caminho para quando você estiver mais experiente com o *framework* e tiver mais conforto para decidir o que é mais útil para você.

Vejamos a migração gerada:

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :full_name
      t.string :email
      t.string :password
      t.string :location
      t.text :bio

      t.timestamps
    end
  end
end
```

Era de se esperar ver SQL, já que estamos tratando de banco de dados, mas ao invés disso vemos Ruby. Isso é uma grande vantagem, já que infelizmente código SQL pode diferenciar entre sistemas de bancos de dados. O `ActiveRecord` já sabe lidar com essas diferenças e portanto apenas escrevemos em uma única linguagem comum.

A migração nada mais é do que uma classe que herda de `ActiveRecord::Migration`. Toda vez que o banco for migrado para uma versão superior, o Rails irá executar o que estiver no método `#change`, ou seja, criando a tabela `users` no banco de dados. Quando o banco for migrado para uma versão inferior, o equivalente do método `create_table` será executado, o `drop_table`, removendo essa tabela.

Dentro do bloco associado ao método `create_table`, os campos serão criados com os tipos que declaramos na execução do gerador. Por fim, o Rails adicionou por

conta própria os campos relacionado à timestamps. São eles o created_at e o updated_at. Esses campos são geridos pelo próprio Rails, ou seja, quando criamos um novo registro no banco, o created_at será automaticamente preenchido. O updated_at é atualizado toda vez que o modelo é gravado novamente no banco, na forma de atualização.

Altere o código da migração para que ela fique da seguinte forma:

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :full_name
      t.string :email
      t.string :password
      t.string :location
      t.text :bio

      t.index :email, unique: true

      t.timestamps
    end
  end
end
```

Com essa alteração, estaremos criando um índice no campo email da tabela users com uma propriedade especial: unicidade. Vejamos o resultado disso em ação. Primeiro, faça:

```
$ rake db:migrate
==  CreateUsers: migrating ========================
-- create_table(:users)
   -> 0.0093s
==  CreateUsers: migrated (0.0140s) ===============
```

Este comando irá criar a tabela no banco de dados e adicionar o índice. Agora vamos ao console do Rails, digitando $ rails console. Esse comando nos leva ao IRB, porém com todo o ambiente do Rails carregado, portanto é possível fazer alterações diretas ao banco de dados.

Novidade Rails 4: índices em migrações

Nas versões anteriores a versão 4, era necessário chamar o método `add_index`, que só podia ser executado fora do contexto do `create_table`:

```
add_index :users, :email, unique: true
```

Com a inclusão do `index` nas tabelas, é possível fazer como fizemos na migração anterior:

```
# ...
  t.index :email, unique: true
# ...
```

Por fim, ainda é possível criar um índice diretamente em chaves estrangeiras (não se preocupe se você não sabe o que é isso, teremos um capítulo só para isso):

```
# ...
  t.references :user, index: true
# ...
```

Atalhos para o comando rails

O comando `rails` possui alguns atalhos para não termos que digitar o nome completo do comando. Veja a lista de atalhos a seguir:

- `rails c` - equivalente ao `rails console`
- `rails s` - equivalente ao `rails server`
- `rails g` - equivalente ao `rails generate`
- `rails db` - equivalente ao `rails dbconsole` (abre o console do cliente de banco de dados, dependendo do qual estiver sendo usado)

```
User.create email: 'admin@example.com'

# A saída é a seguinte (quebrada em linhas para facilitar a leitura):
# => #<User id: 1,
#           full_name: nil,
#           email: "admin@example.com",
#           password: nil,
#           location: nil,
#           bio: nil,
#           created_at: "2012-11-29 07:22:45",
#           updated_at: "2012-11-29 07:22:45">
```

Podemos observar algumas coisas importantes neste exemplo. Primeiro é que o Rails criou o campo `id` mesmo sem qualquer declaração na migração. Esse campo é uma chave primária, ou seja, um identificador único de uma entrada na tabela. Este campo é muito comum no desenvolvimento de projetos, portanto o Rails sempre criará este campo para você.

Podemos ver vários atributos como `nil`, o que é esperado, pois não damos nenhum valor a eles. Porém, podemos ver que o Rails automaticamente populou os campos `created_at` e `updated_at`. Esses campos também são automaticamente gerenciados pelo Rails, **caso** os campos existam. Porém, a existência desses campos não é obrigatória nada acontecerá se eles não existirem.

Dica: console do Rails em modo sandbox

É muito comum fazermos alguns testes manualmente, criando, alterando ou deletando objetos do banco de dados. Para que nossos testes não fiquem poluindo o sistema, é possível iniciar o console usando uma chave para `sandbox`: `rails console --sandbox`. Quando você iniciar a sessão do console, o Rails irá emitir um início de transação no banco por todo o tempo. Quando você sair, um `ROLLBACK` será enviado ao banco, desfazendo todas as alterações.

Vejamos o modelo usuário, localizado em `app/models/user.rb`:

```
class User < ActiveRecord::Base
end
```

Se você observar no exemplo que executamos no *console*, poderá ver que muitos campos ficaram em branco. Para evitar cadastros muito ruins ou até mesmo acidentes, vamos colocar algumas validações, garantindo que o usuário coloque todos os dados necessários.

4.2 Evite dados errados. Faça validações

O `ActiveRecord` nos disponibiliza diversos tipos de validações já prontas para grande parte das situações comuns, tais como validar presença, formato, inclusão em uma lista ("sexo" em "masculino" ou "feminino", por exemplo), entre vários outros, além de disponibilizar ferramentas para criarmos as nossas próprias validações.

Tudo isso é feito de forma bastante conveniente, usando *class macros*. Vamos então validar:

- a presença do email, nome completo, localização e senha;
- a confirmação de senha, ou seja, o usuário precisa preencher um campo contendo a senha e outro para verificar se o usuário não cometeu erros de digitação;
- mínimo de 30 caracteres para a bio.

Vamos primeiro adicionar as validações de presença, informações obrigatórias no formulário:

```
class User < ActiveRecord::Base
  validates_presence_of :email, :full_name, :location, :password
end
```

Salve o arquivo, pois vamos experimentar essas validações no console (basta executar novamente o comando `rails console`):

```
user = User.new
user.valid? # => false
user.save   # => false

user = User.new
user.email = 'joao@example.com'
user.full_name = 'João'
user.location = 'São Paulo, Brasil'
user.password = 'segredo'

user.valid? # => true
user.save   # => true
```

Quando executamos o método `#valid?` no modelo, o `ActiveRecord` irá executar a série de validações que o modelo tiver e retornar o resultado. Se o objeto estiver válido, retornará `true`, e `false` caso contrário, como é de se esperar.

O `#save` irá executar todas as validações (via o método `#valid?`) e se o modelo estiver válido, tentará salvar o modelo no banco de dados e retornar verdadeiro. Porém, se algum problema ocorrer no processo, seja validação ou falha ao salvar o objeto no banco de dados, o método `#save` retornará falso.

> **De onde surgiram os métodos acessores?**
>
> Quando criamos nossos modelos e os fazemos herdar de `ActiveRecord::Base`, automaticamente, o Rails disponibiliza os métodos de leitura e escrita de todas as colunas que a tabela possui.

Bom, já temos uma ideia de como funcionam as validações, então vamos adicionar uma nova:

```
class User < ActiveRecord::Base
  validates_presence_of :email, :full_name, :location, :password
```

```
  validates_confirmation_of :password
end
```

Dica: recarregando as classes

Se você alterar o modelo e não reiniciar sua sessão do console do Rails, você não vai conseguir interagir com as suas novas alterações. Para não haver a necessidade de sair e entrar novamente no console, basta executar o comando `reload!` que o Rails irá recarregar tudo o que estiver dentro da pasta `app`.

O funcionamento do `validates_confirmation_of` é bem interessante. A partir do momento que adicionamos essa validação, para salvar um objeto no banco de dados, é necessário usar um novo atributo "virtual", ou seja, um atributo que não existe no banco de dados, chamado `password_confirmation`. Se ele não estiver igual ao campo de senha (`password`), o modelo não pode ser gravado no banco.

Atributos virtuais

Como o Rails interage com os métodos padrão de leitura e escrita do Ruby, é possível criar atributos que não são persistidos no banco de dados, mas que interagem com o Rails. Por exemplo, o próprio `ActiveRecord::Base` cria um atributo virtual chamado `password_confirmation` e realiza validações em `password` com este novo atributo.

Essa técnica é bastante útil para simplificar a interface externa de uma classe, de modo a esconder detalhes de implementação para outros elementos.

Com o mesmo console aberto, podemos executar:

```
reload!
# Reloading...
# => true

user = User.new
```

```
user.email = 'joao@example.com'
user.full_name = 'João'
user.location = 'São Paulo, Brasil'
user.password = 'segredo'
user.password_confirmation = 'errei_o_segredo'

user.valid? # => false
user.errors.messages
# => {:password=>["doesn't match confirmation"]}
```

Quando a validação é executada (tanto pelo método `#valid?` quanto pelo `#save`), o `ActiveRecord` popula um atributo especial no modelo, chamado `errors`. Com ele é possível verificar, em mensagens legíveis, quais foram os erros de validação.

Agora vamos a última validação, o tamanho da bio:

```
class User < ActiveRecord::Base
  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
  validates_length_of :bio, minimum: 30, allow_blank: false
end
```

O `validates_length_of` faz diversos tipos de validação com tamanho de texto. Nesse caso, aplicamos ao atributo "bio" apenas duas restrições: o tamanho mínimo de 30 caracteres e ele não pode ser em branco. Porém, o `validates_length_of` aceita outras opções, entre elas:

- `:maximum`: Limita o tamanho máximo;
- `:in`: Ao invés de passar `:minimum` e `:maximum`, basta fazer, por exemplo, `in: 5..10` para validar o mínimo de 5 e o máximo de 10 caracteres;
- `:is`: Limita o tamanho exato do texto;
- `:allow_blank`: Permite que o atributo fique em branco, e, quando presente, respeite as condições de tamanho;

Existem algumas outras opções menos usadas. Para saber quais são, recomendo olhar a documentação oficial do Rails.

O `ActiveRecord` ainda possui diversas outras validações que não usamos nessa classe, mas que são bastante úteis. Exemplos:

- `validates_format_of`: Valida o formato de um texto com uma expressão regular;
- `validates_inclusion_of`: Valida a inclusão de um elemento em um enumerável, ou seja, um número em um *range* ou um texto dentre uma lista de opções;
- `validates_numericality_of`: Valida se o atributo passado é realmente um número, com algumas opções interessantes, por exemplo, `:less_than`, `:greater_than`, entre outros.

Além das validações citadas, ainda é possível criar validações customizadas, que é o que vamos fazer agora. Vamos validar o formato do email, pois é possível colocar qualquer coisa, mesmo que não seja um e-mail válido. Observe que é possível implementar essa mesma validação com o `validates_format_of`, mas vamos usar uma validação customizada para entender como ela funciona.

Para isso, vamos usar uma validação simples. Ela não é nem de longe ideal e não é compatível com o RFC de e-mails (veja em http://bit.ly/validacao-email uma expressão regular compatível, se estiver curioso), mas é suficiente para informar ao usuário que algo está errado. Já que vamos enviar um email de confirmação para o usuário poder usar o sistema, não é necessária tanta formalidade.

Vamos usar uma expressão regular extraída da biblioteca Devise (http://github.com/plataformatec/devise) , uma biblioteca complexa de autorização de usuários. Apesar de completa, a biblioteca não é recomendada para quem está começando com Rails, portanto não vamos usá-la no Colcho.net.

```
class User < ActiveRecord::Base
  EMAIL_REGEXP = /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/

  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
  validates_length_of :bio, minimum: 30, allow_blank: false

  validate :email_format

  private

  # Essa validação pode ser representada da seguinte forma:
  # validates_format_of :email, with: EMAIL_REGEXP
```

```
  def email_format
    errors.add(:email, :invalid) unless email.match(EMAIL_REGEXP)
  end
end
```

A *class macro* `validate` aceita um símbolo com o nome do método a ser chamado ou um bloco. O efeito seria o mesmo se tivéssemos feito da seguinte forma:

```
validate do
  errors.add(:email, :invalid) unless email.match(EMAIL_REGEXP)
end
```

Observe a forma que anotamos que um atributo é inválido: chamamos o método `add` (adicionar) no objeto `errors` com o nome do atributo e, em seguida, o tipo do erro. O tipo do erro é usado para determinar a mensagem que será exibida ao usuário e pode ser qualquer coisa que você quiser. Usamos `:invalid` pois o Rails já usa e sabe traduzir. Veja a lista a seguir, contendo as mensagens mais interessantes, retirada do arquivo de traduções do Rails (https://github.com/svenfuchs/rails-i18n/blob/master/rails/locale/en.yml#L100)

- `accepted`: must be accepted (precisa ser aceito)
- `blank`: can't be blank (não pode ficar em branco)
- `confirmation`: doesn't match confirmation (não é igual a confirmação)
- `empty`: can't be empty (não pode ficar vazio)
- `equal_to`: must be equal to %{count} (precisa ser igual a `count`)
- `exclusion`: is reserved (é reservado)
- `greater_than`: must be greater than %{count} (precisa ser maior que `count`)
- `greater_than_or_equal_to`: must be greater than or equal to %{count} (maior ou igual que `count`)
- `inclusion`: is not included in the list (não está na lista)
- `invalid`: is invalid (é invalido)
- `less_than`: must be less than %{count} (precisa ser menor que `count`)

- `less_than_or_equal_to`: must be less than or equal to %{count} (menor ou igual que `count`)
- `not_a_number`: is not a number (não é um número)

No capítulo 7 "Faça sua aplicação falar várias línguas", vamos ver mais detalhes sobre o sistema de internacionalização do Rails.

Vamos também criar uma validação de unicidade para emails, ou seja, emails cadastrados não poderão existir previamente no site. É importante ressaltar que o `validates_uniqueness_of` possui um problema: primeiro o Rails verifica a existência do email a ser cadastrado e **depois** cria o modelo no banco, se o resto estiver OK. É possível que, entre a verificação e a criação, o email seja criado no banco e o Rails tentará criar o modelo de qualquer forma. É por isso que criamos a validação de unicidade também no banco de dados, e estamos criando essa validação no modelo apenas para *feedback* ao usuário, resultando no seguinte código:

```
class User < ActiveRecord::Base
  EMAIL_REGEXP = /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/

  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
  validates_length_of :bio, minimum: 30, allow_blank: false
  validates_uniqueness_of :email

  # Essa validação pode ser representada da seguinte forma:
  # validates_format_of :email, with: EMAIL_REGEXP
  validate do
    errors.add(:email, :invalid) unless email.match(EMAIL_REGEXP)
  end
end
```

Sintaxe alternativa para validações

Existe uma outra sintaxe para validações. Veja as duas validações a seguir:

```
validates :email, presence: true
```

```
validates_presence_of :email
```

Ambas possuem o mesmo comportamento, porém a sintaxe usando a *class macro* `validates` foca no atributo a ser validado, de forma que você possa adicionar diversas validações de uma vez. O `validates_presence_of`, em contrapartida, foca na validação em si, podendo passar diversos atributos de uma vez. Veja outro exemplo:

```
validates :email, presence: true,
                  format: { with: /\A[^@]+@[^@\.]+\z/ },
                  uniqueness: true
```

Ainda é possível colocar múltiplos atributos na validação:

```
validates :email, :full_name, :location, presence: true
```

Como não há diferença de comportamento, você pode converter as validações que fizemos com a sintaxe tradicional para a sintaxe alternativa e verificar qual você mais gosta.

Nosso modelo usuário está bom o suficiente para prosseguir. Vamos agora começar a fazer o fluxo de cadastro, criando o controle e customizando as páginas.

Capítulo 5

Tratando as requisições Web

Eu não desanimo, pois cada tentativa incorreta descartada é mais um passo a frente.
– Thomas A. Edison

5.1 Roteie as requisições para o controle

Vamos começar a ligar as outras partes do sistema com o modelo que criamos no capítulo anterior. Vamos fazer tudo com o mínimo uso de geradores do Rails, para entendermos cada passo do projeto, diferente do que fizemos com o modelo "quarto".

Antes de prosseguirmos, temos que criar algum código para que o roteador do Rails saiba interpretar requisições em uma certa URL. Vamos alterar o roteador de maneira a adicionar as seguintes ações para o recurso usuário:

- **index** - Lista todas as entradas do recurso;
- **show** - Exibe uma entrada específica do recurso;
- **new** - Página para entrar com os dados para uma nova entrada;

- **create** - Ação de criar uma nova entrada;
- **edit** - Página para editar uma entrada já existente;
- **update** - Ação de atualização de uma entrada existente;
- **destroy** - Remoção de uma entrada existente;

Uma vez que indicarmos no roteador essas rotas, precisamos também indicar qual controle irá responder pelas requisições. Para fazer isso, vamos editar o roteador (`config/routes.rb`) da seguinte forma:

```
Colchonet::Application.routes.draw do
  resources :rooms
  resources :users

  # Aqui estarão vários comentários gerados pelo Rails para
  # te ajudar a lembrar e entender como funciona
  # o roteador.
end
```

Agora, com o servidor do Rails em execução (lembre-se, basta executar `rails server` na pasta do projeto), abra o browser e acesse o endereço para o nosso recurso: http://localhost:3000/users/new.

Você vai encontrar uma tela de erro:

Routing Error

uninitialized constant UsersController

Rails.root: /Users/vinibaggio/Projects/colcho.net

Application Trace | Framework Trace | Full Trace

Routes

Routes match in priority from top to bottom

Helper	HTTP Verb	Path	Controller#Action
Path / Url		Path Match	
rooms_path	GET	/rooms(.:format)	rooms#index
	POST	/rooms(.:format)	rooms#create
new_room_path	GET	/rooms/new(.:format)	rooms#new
edit_room_path	GET	/rooms/:id/edit(.:format)	rooms#edit
room_path	GET	/rooms/:id(.:format)	rooms#show
	PATCH	/rooms/:id(.:format)	rooms#update
	PUT	/rooms/:id(.:format)	rooms#update
	DELETE	/rooms/:id(.:format)	rooms#destroy
users_path	GET	/users(.:format)	users#index
	POST	/users(.:format)	users#create
new_user_path	GET	/users/new(.:format)	users#new
edit_user_path	GET	/users/:id/edit(.:format)	users#edit
user_path	GET	/users/:id(.:format)	users#show
	PATCH	/users/:id(.:format)	users#update
	PUT	/users/:id(.:format)	users#update
	DELETE	/users/:id(.:format)	users#destroy

Figura 5.1: Erro: Routing error - Erro de roteamento

O *Routing error* (erro de roteamento) é um erro comum. O roteador do Rails tenta passar a execução de requisições de um recurso para o seu controle correspondente. No caso do modelo usuário, o Rails tentará passar requisições em rotas que iniciam com `/users` para o controle `UsersController`, que ainda não existe.

Quando especificamos `resources :rooms`, o roteador espera que exista um controle `UsersController` que responda às sete ações que vimos anteriormente, no início desse capítulo. O mesmo acontece com o recurso quartos (`resources :rooms`). Como usamos o gerador para o recurso quarto, o controle foi criado automaticamente. No caso de usuários, vamos criá-lo manualmente, em seguida.

Novidade Rails 4: Telas de erros

Uma novidade do Rails 4 é que as telas de erro de roteamento são muito mais informativas. No final da página, há uma lista de rotas que sua aplicação dá suporte ordenadas por prioridade. Ou seja, caso alguma rota que você tenha entre em conflito com outra, a rota que tiver maior prioridade será executada.

Os controles vivem na pasta `app/controllers`, portanto crie o arquivo `users_controller.rb` de acordo com o seguinte:

```
class UsersController < ApplicationController
end
```

Como você pode observar, um controle é uma classe Ruby comum, que herda da classe `ApplicationController`. O `ApplicationController` é um controle "abstrato", ou seja, não terá uma rota que aponta diretamente para ela, porém ela define características globais em sua aplicação. Essa classe é gerada quando a aplicação é criada, ou seja, quando executamos, na linha de comando, o comando `rails new` e portanto, vive em seu projeto. Ao abrir o arquivo `app/controllers/application_controller.rb`, temos o código a seguir, gerado pelo Rails:

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
end
```

O `ApplicationController` herda, por sua vez, do `ActionController::Base`, um componente interno do Rails. Nessa classe há uma *class macro* bastante importante, `protect_from_forgery`. Essa macro faz com que todos os controles da aplicação exijam uma chave de autenticação em ações de alteração de dados (*create*, *update* e *destroy*) de modo a evitar ataques de falsificação de requisição (*Request Forgery*). Portanto, é bastante importante deixá-la ativada sempre que possível.

Falsificação de requisição

Ataques de falsificação de requisição entre sites (ou *cross-site request forgery* - CSRF) consistem em enviar uma requisição de um site a outro fazendo uma ação que o usuário não desejou. Por exemplo, imagine que no colcho.net, para você deletar sua conta, haja o endereço hipotético http://colcho.net/usuarios/deletar_conta . Um usuário malicioso pode colocar em um site qualquer uma imagem com a seguinte tag:

`<img src="http://colcho.net/usuarios/deletar_conta" />`

Ao entrar nesse site, o browser irá tentar baixar o conteúdo dessa imagem, que na verdade é a ação para deletar sua conta e **bum**!, sua conta foi deletada. Para proteger seu site contra isso, o Rails faz um mecanismo tal que, para cada ação de modificação de estado, é necessário enviar uma chave de segurança gerada aleatoriamente.

Prosseguindo com o projeto, vamos adicionar uma ação ao controle de usuários. Essa ação será a ação de cadastro, ou seja, a criação de um novo objeto usuário. Para isso, criaremos a ação `new`, no `UsersController` (`app/controllers/users_controller.rb`):

```
class UsersController < ApplicationController
  def new
  end
end
```

Por agora não faremos nada nessa ação, portanto o método `new` fica vazio.

Ao acessarmos http://localhost:3000/users/new , um erro diferente é apresentado: **Template is missing**, ou "template ausente". A ação é encontrada, porém o Rails não consegue encontrar a apresentação daquela ação:

Template is missing

Missing template users/new, application/new with {:locale=> [:en], :formats=>[:html], :handlers=>[:erb, :builder, :raw, :ruby, :jbuilder, :coffee]}. Searched in: * "/Users/vinibaggio/Projects/colcho.net/app/views"

Figura 5.2: Erro: Template is missing - template ausente

Os templates ficam na pasta `app/views`. Dentro dela, existe uma hierarquia. Mais uma vez o Rails adota uma convenção. Da mesma maneira que o nome do controle é derivado do nome do recurso, o nome da pasta é derivado do nome do controle. Por exemplo, se tivéssemos o controle `UserSessionsController`, os templates deste controle serão procurados na pasta `app/views/user_sessions`. Dessa forma, os templates para o controle `UsersController` serão procurados na pasta `app/views/users`.

Note que na pasta `app/views` existe também a pasta `layouts`. Essa pasta é uma pasta especial. Os templates presentes nessa pasta são usados por um ou mais controles e servem como a base em que vamos injetar o template de cada ação. Para entender melhor, observe a figura 5.3.

```
<!DOCTYPE html>                app/views/layouts/application.html.erb
<html>
<head>
  <title>Colchonet</title>
  <%= stylesheet_link_tag "application",
                          media: "all",
                         "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application",
                             "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>

</head>
<body>

                                   app/views/users/new.html.erb

</body>
</html>
```

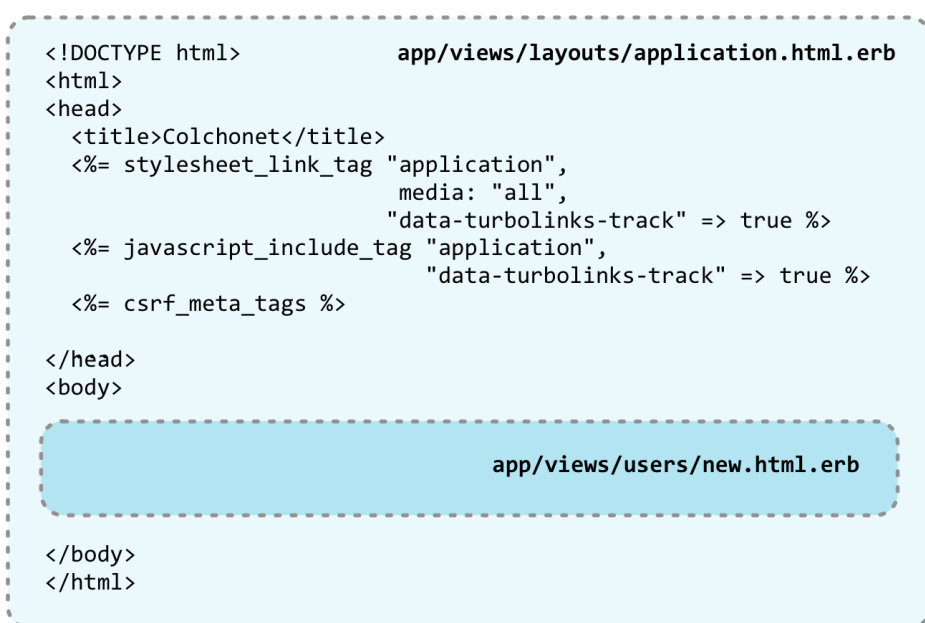



Figura 5.3: Composição de página usando layouts

O uso de layouts dessa forma é bastante prático. Não há a necessidade de repetir diversos comandos e tags (evitando também um tedioso trabalho de busca/substituição quando você faz alguma alteração), basta escrever o HTML específico da sua página que será injetado no código do layout mais genérico para a sua aplicação.

O que difere layouts de templates de ações é a execução do comando `yield`. O `yield` diz ao Rails que, naquele exato ponto, deverá ser injetado um outro template. Veja o exemplo a seguir:

`application.html.erb`:

```
<html>
  <body>
    <%= yield %>
  </body>
</html>
```

Agora imagine que o template da ação `new` do controle `UsersController` seja da seguinte forma:

```
<h1>Cadastro</h1>
```

Quando executarmos, no controle, o seguinte código:

```
render "new", layout: "application"
```

O Rails irá renderizar para o usuário o seguinte resultado final:

```
<html>
  <body>
    <h1>Cadastro</h1>
  </body>
</html>
```

É possível também ter mais de um layout em um sistema. Isso é útil, por exemplo, quando há diversas "personas" em um site, como um administrador e um usuário comum, ou um lojista e o cliente.

Personas

Persona é um personagem fictício de um possível usuário do seu site. É importante pensarmos em como cada persona vai interagir com o sistema que estamos elaborando. Assim, é mais fácil ter ideias de como melhorar a experiência do usuário.

No caso do colcho.net, temos duas personas: o "hospedado", usuário que procura lugar para dormir e o "anfitrião", usuário que tem um lugar sobrando.

Vamos agora criar o template para a ação *new*. Para isso, crie a pasta `users` em `app/views` e crie o arquivo `app/views/users/new.html.erb`. O nome do arquivo possui três significados: ação *new*, que vai ser pré-processado pelo ERB, resultando em um arquivo `html`:

```
<h1>Cadastro</h1>
```

Por enquanto esse template está muito sem graça. Vamos usar o controle `UsersController` para preparar os dados para que possamos criar o formulário de cadastro.

5.2 Integre o controle e a apresentação

Vamos voltar ao `UsersController` para criar a tela de cadastro de novo usuário:

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end
end
```

Duas coisas acontecem no trecho de código anterior: a primeira é que usamos o método `.new` no modelo `User`. Já vimos esse método anteriormente, ele cria uma nova instância do modelo `User`, sem gravar no banco de dados. Em seguida, armazenamos o novo objeto em uma variável de instância, `@user`. O controle compartilha todas as variáveis de instância com o template, ou seja, toda variável com `@` estará disponível no template renderizado pela ação.

Note que nunca especificamos o template a ser renderizado. Por padrão, quando não especificamos o template, o Rails irá buscar pelo nome equivalente da ação na pasta de templates do controle. Como a ação é `new`, o Rails irá buscar por `app/views/users/new.html.erb`.

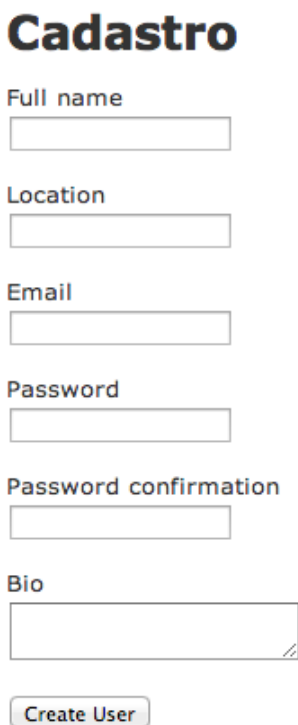
Isso significa que já podemos acessar esse objeto no template. Usaremos ele para construir o formulário. Esse formulário terá vários campos para preencher o objeto. Abra o arquivo `app/views/users/new.html.erb` e altere para que ele fique da seguinte forma:

```
<h1>Cadastro</h1>

<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %><br />
    <%= f.text_field :full_name %>
  </p>
  <p>
    <%= f.label :location %><br />
    <%= f.text_field :location %>
  </p>
  <p>
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </p>
```

```
  <p>
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %><br />
    <%= f.text_area :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Esse código que criamos é um exemplo de como é um template ERB: trechos de HTML envolto de código Ruby. Em ERB, todo código Ruby que não terá seu conteúdo impresso no HTML resultante deve ser envolto de `<<% %<>`. Caso você queira que o código seja impresso no HTML resultante, temos que envolver o código Ruby com `<%= %>` (observe o `=`).

Você pode ver também que este código faz uso extensivo de métodos auxiliares do Rails. Nesse caso, utilizamos alguns métodos, como o `text_field` e `password_field`, para construir o formulário:

- `form_for` - Inicia a construção de um formulário para um modelo;
- `label` - *Label* (etiquetas) correspondente a um campo do formulário;
- `text_field` - Campo simples de texto;
- `password_field` - Campo mascarado de senha;
- `text_area` - Área grande de texto, para escrevermos a bio;
- `submit` - Botão de Submit (Enviar), para enviar os dados ao servidor;

O `form_for` é o que chamamos de construtor de formulários para modelos. Esse método é bastante integrado com todas as camadas do Rails. Por exemplo, ele

sabe construir exatamente a rota que deverá ser chamada quando enviamos o formulário (via botão "Submit"): ele usa o método `#new_record?` no modelo para determinar se o objeto está gravado no banco de dados, e portanto, sabe diferenciar se deve rotear o usuário para a ação `create` ou `update`.

Ele sabe também que o objeto possui atributos (experimente mudar um dos campos do formulário para algum atributo qualquer, não existente no modelo e veja o que acontece) e portanto faz o mapeamento correto para que o atributo seja facilmente usado no controle, para gravar o objeto. Além disso, se o objeto já existir no banco de dados, ele irá popular as caixas de texto com os valores associados a cada atributo.

Vejamos o resultado deste template:

Figura 5.4: Formulário de cadastro de usuário

Ao preencher o formulário e clicar em "Create User" (criar usuário), vamos ser apresentados ao erro *Unknown action*, ou ação desconhecida, *create*. Isso significa

que o formulário enviou os dados para uma ação que ainda não existe. Pois vamos criá-la. Nessa ação, temos que obter os dados do formulário e gravá-los como um novo objeto no banco de dados. Se o objeto for gravado com sucesso, vamos apresentar o usuário com uma mensagem de sucesso, e, caso contrário, apresentamos o formulário novamente, anotado com erros, para que o usuário possa corrigi-los e enviar os novos dados.

A ação *create* possuirá a seguinte lógica:

- Cria novo usuário baseado nos dados vindos do formulário, via o método `params`;
- Se o usuário foi salvo com sucesso (`#save` retorna sempre `true` ou `false`), redireciona ele para a página de seu perfil e exibe uma mensagem de sucesso;
- Caso não seja possível salvar, renderiza o formulário novamente.

Abra novamente o `UsersController` (`app/controllers/users_controller.rb`) e deixe-o da seguinte maneira:

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new(params[:user])
    if @user.save
      redirect_to @user,
                  notice: 'Cadastro criado com sucesso!'
    else
      render action: :new
    end
  end
end
```

Vamos entrar em detalhe em cada linha:

```
@user = User.new(params[:user])
```

O método `params` (lembre-se que em Ruby, para chamar métodos, não precisamos de parênteses) retorna um hash com todos os parâmetros enviados pelo usuário,

seja via formulário (**POST**) ou via *query string* (parâmetros pela URL, por exemplo http://google.com?q=hello, via **GET**). Quando usamos os métodos auxiliares do Rails para formulários, o `params` se parece com o seguinte:

```
{
  "authenticity_token"=>"ofiNLJQUjL/p5vX8zOcy+N5aE9htJDIAUk=",
  "user"=> {
    "full_name"=>"Vinicius",
    "location"=>"San Francisco, CA",
    "email"=>"vinibaggio@example.com",
    "password"=>"segredo",
    "password_confirmation"=>"segredo",
    "bio"=>"Olá, tudo bom? Meu nome é Vinícius."
  },
  "commit"=>"Create User",
  "action"=>"create",
  "controller"=>"users"
}
```

Filtro de atributos

Se você observar o log do servidor, notará algumas diferenças entre os atributos que vimos anteriormente e o que de fato é passado no *hash* `params`. A principal diferença é que o Rails, por padrão, filtra qualquer parâmetro que contenha a palavra "password" no nome. Isso é bastante importante pois, se algum usuário obter os logs do servidor, ele não será capaz de recuperar nenhuma informação sensível.

Para adicionar ou mudar esta lista de atributos, basta alterar o arquivo `config/initializers/filter_parameter_logging.rb`:

```
# Be sure to restart your server when you modify this file.

# Configure sensitive parameters which will be
# filtered from the log file.
Rails.application.config.filter_parameters += [
  :password,
  :password_confirmation
]
```

Você pode ver que todos os campos do formulário vieram dentro de uma hash só, a hash `users`. É esse conteúdo que passamos para o método `.new` do modelo.

Chave da hash em string ou símbolo?

O hash `params` é, normalmente, uma coisa confusa para iniciantes pois você pode acessá-lo usando chaves como símbolos ou strings. Isso significa que, por exemplo, `params["user"]` e `params[:user]` produzem o mesmo resultado.

Isso acontece pois o Rails cria um tipo especial de hashes, chamado `HashWithIndifferentAccess`, que pode ser usado da mesma maneira que um hash comum.

O método `.new` aceita um hash contendo todos os atributos a serem associados. Esse é um conceito chamado *mass-assignment*, ou seja, associação em massa. É muito mais prático do que ficar chamando cada método individualmente dessa maneira. Porém, essa associação em massa pode gerar problemas de segurança. Veremos esse assunto logo em seguida, na seção

Continuando, com o objeto populado de atributos que vieram do formulário, tentaremos salvá-lo no banco de dados. Se tudo der certo, ou seja, `#save` retornar `true`, as linhas a seguir serão executadas:

```
redirect_to @user,
          notice: 'Cadastro criado com sucesso!'
```

O método `redirect_to` envia ao browser do usuário um código de resposta "302" que significa "*Moved Temporarily*", dizendo ao navegador que ele deve ir para outro endereço. No conteúdo da resposta, um endereço é informado, fazendo o browser seguir esta nova localização. Por sua vez, passamos o modelo `@user` para o `redirect_to`, que, como o `form_for` sabe montar a URL correta a partir do objeto. Neste caso, o Rails irá redirecionar para a ação *show* com o ID do objeto `@user`.

O segundo parâmetro do método `redirect_to` é um hash de opções. Nesse caso, estamos usando um atalho para escrever uma mensagem via `flash`.

O `flash` é fundamentalmente um hash, que nesse caso estamos escrevendo na chave `:notice`. Portanto a linha poderia ser reescrita da seguinte forma:

```
flash[:notice] = 'Cadastro criado com sucesso!'
redirect_to @user
```

Uma característica importante do `flash` é que seu conteúdo é guardado na sessão do usuário e o conteúdo é descartado em toda requisição. Quando você escreve no `flash`, o conteúdo só estará disponível na próxima requisição. Por isso, é normalmente utilizado em conjunto com redireções.

Finalmente, se o objeto não pôde ser salvo, executamos:

```
render action: :new
```

Isso porquê a ação *create* não possui um template/apresentação para si, então reaproveitamos o template do *new* para apresentar novamente o formulário e o usuário ter a oportunidade de corrigir os dados. Como já obtivemos o modelo do usuário na variável `@user`, ela já estará disponível no template. Note que chamar o método `render` não redireciona o usuário e também não executa a ação indicada por completo, apenas usa o seu template.

REDIRECT_TO OU RENDER?

É comum em ações *create* e *update* redirecionar o usuário para uma outra página quando tudo está certo. Porém, no caso de erros no objeto não é recomendado redirecionar. A razão é que, ao redirecionar, perdemos todos os parâmetros que o usuário enviou e também as mensagens de erro de validação quando executamos o método `#save`. Para essas situações, devemos usar o `#render`, que não causa o redirecionamento.

5.3 Controle o *mass-assignment*

Agora que entendemos bem o que a ação *create* faz, vamos executá-la! Vá ao formulário de cadastro (http://localhost:3000/users/new) e preencha o formulário. Ao clicar em 'Create User'...

```
ActiveModel::ForbiddenAttributesError in UsersController#create
```

A exceção `ForbiddenAttributesError` ocorre quando usamos o `params` para construir objetos sem listarmos, manualmente, todos atributos que podem ser

atualizados a partir do formulário. Esta situação é na verdade uma proteção para o seu site. Imagine a situação hipotética em que tivéssemos o campo `admin` no modelo `User`, que é um campo booleano (ou seja, aceita apenas valores `true` ou `false`).

Mesmo sem o campo `admin` existir no formulário, um usuário mal intencionado pode forjar uma requisição da seguinte forma:

```
{
  "user" => {
    "full_name"=>"Pirata Malandro",
    "location"=>"Caribe",
    "email"=>"malandro@example.com",
    "password"=>"123",
    "password_confirmation"=>"123",
    "bio"=>"Rárr! Vou adquirir acesso de admin!",
    "admin"=>"1"
  }
}
```

Mesmo seu formulário não tendo o campo `admin`, o modelo `User` iria marcá-lo como `true` e pronto, facilmente um usuário pode forjar dados em seu sistema sem você perceber.

O Rails 4 introduziu um novo mecanismo para determinar quais campos são seguros, e para isso precisamos alterar o controle.

Atualizando do Rails 3

Se você já conhece o Rails 3, vai notar que há uma grande diferença: os atributos não são mais marcados como seguros no modelo, mas sim no controle. O grande benefício é que mais de um controle pode alterar um modelo, dependendo da regra de negócio daquele controle. Por exemplo, um administrador poderia alterar mais campos no painel de administração do que um usuário comum.

Outra grande vantagem é a possibilidade de alterar modelos mais naturalmente em processos que não envolvam controles de requisições web, como processamento de dados em background.

Se você quiser ainda usar o mesmo mecanismo do Rails 3 em um projeto Rails 4, basta adicionar a *gem* `protected_attributes` no `Gemfile` do seu projeto. Por sua vez, se quiser usar o novo mecanismo em projetos Rails 3, basta instalar a usar a gem `strong_parameters`, lembrando que não há necessidade de instalá-la em projetos Rails 4.

Para criar uma lista de atributos permitidos, vamos criar um método que irá garantir que os parâmetros incluem apenas dados permitidos:

```
class UsersController < ApplicationController

  # ...
  private

  def user_params
    # Os "pontos" no final da linha não são opcionais!
    params.
      require(:user).
      permit(:email, :full_name, :location, :password,
             :password_confirmation, :bio)
  end
end
```

Muita coisa está acontecendo neste método. Primeiro, chamamos o método `require` no objeto retornado pelo método `params` (note o "ponto" no final da linha: apenas estamos quebrando a linha para melhorar legibilidade).

Este método irá verificar pela presença da chave `:user` nos parâmetros vindos do usuário. Se essa chave não existir, o método irá disparar a exceção `ActionController::ParameterMissing`. Se existir, os parâmetros presentes a partir dessa chave serão retornados.

Na linha seguinte, usamos o retorno do método `require` para fazer a verificação dos atributos. Mais uma vez estamos usando o "ponto" no final da linha, de modo a encadear chamadas de métodos. O `permit` tem um papel diferente: ele irá retornar um hash com apenas os parâmetros listados, excluindo os outros atributos não especificados.

Veja o exemplo a seguir, executados em uma mesma requisição. O método `params[:user]` retorna todos os parâmetros dentro da chave `:user`, sem nenhum filtro de *mass assignment* aplicado a ele. O método `params.require(:user).permit(:email, :full_name)` irá filtrar todos os atributos com exceção de `:email` e `:full_name`. No último exemplo, colocamos a `:bio` a mais.

```
params[:user]
# {
#   "full_name"=>"Teste",
#   "location"=>"teste",
#   "email"=>"vinibaggio@gmail.com",
#   "password"=>"123qwe",
#   "password_confirmation"=>"123qwe",
#   "bio"=>"Este é um teste de bio para o Colcho.net"
# }

params.
  require(:user).
  permit(:email, :full_name)
# {
#   "email"=>"vinibaggio@gmail.com",
#   "full_name"=>"Teste"
# }

params.
  require(:user).
  permit(:email, :full_name, :bio)
# {
#   "email"=>"vinibaggio@gmail.com",
#   "full_name"=>"Teste",
```

```
#   "bio"=>"Este é um teste de bio para o Colcho.net"
# }
```

Exibindo exceções para o usuário

Enquanto desenvolvemos o colcho.net, encontramos várias exceções, como `RoutingError`, ou até mesmo o `ActionController::ParameterMissing`. Pode parecer estranho exibir essas exceções, porém o Rails apenas faz isso em ambiente de desenvolvimento, para facilitar a depuração de bugs.

No ambiente de produção, essas exceções mapeiam para erros específicos, como "404 Not Found", "400 Bad Request" e, se alguma exceção não for conhecida pelo Rails ou não for tratada pelo desenvolvedor, o Rails irá apresentar o erro "500 Internal Server Error". Veja uma lista abaixo das exceções e a qual resposta elas são mapeadas:

- **ActionController::RoutingError** - "404 Not Found"
- **AbstractController::ActionNotFound** - "404 Not Found"
- **ActionController::MethodNotAllowed** - "405 Method Not Allowed"
- **ActionController::NotImplemented** - "501 Not Implemented"
- **ActionController::UnknownFormat** - "406 Not Acceptable"
- **ActionController::InvalidAuthenticityToken** - "422 Unprocessable Entity"
- **ActionController::BadRequest** - "400 Bad Request"
- **ActiveRecord::RecordNotFound** - "404 Not Found"
- **ActiveRecord::RecordInvalid** - "422 Unprocessable Entity"

Para aplicar a proteção de *mass assignment*, basta alterar a linha que usamos `params[:user]` e usar o método que acabamos de criar, o `user_params`. O resultado final do controle deverá ficar da seguinte maneira:

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def create
    # Lembre-se de alterar a linha a seguir!
    # mude de params[:user] para user_params.
    @user = User.new(user_params)
    if @user.save
      redirect_to @user,
                  notice: 'Cadastro criado com sucesso!'
    else
      render action: :new
    end
  end

  private

  def user_params
    params.
      require(:user).
      permit(:email, :full_name, :location, :password,
             :password_confirmation, :bio)
  end
end
```

Depois dessas alterações, reenvie o formulário, com todos os dados. Se não houveram erros no formulário, um novo erro será exibido: *Unknown action* (ação desconhecida), *show*. Apesar do erro, o caminho tomado no código foi o caminho de sucesso e, portanto, o usuário é redirecionado para a ação *show*. Para comprovar que o objeto foi gravado, vá ao console do Rails:

```
# Busca o último objeto no banco de dados (via id)
User.last
#<User id: 1, full_name: "Vinicius", ...>
```

5.4 Exibição do perfil do usuário

Agora vamos criar a ação e o template de *show*, para não haver mais erros. Primeiro, precisamos buscar o objeto no banco de dados, de modo a popular os

dados adequados na tela. Para isso, vamos criar um novo método no controle `UsersController`, que buscará o objeto no banco de dados e irá montar o template com os dados desse objeto.

Em projetos Rails, usamos o ID do objeto na rota. Ou seja, quando quisermos renderizar a ação `show` para o objeto de ID 1, basta seguirmos a rota `/users/1`. Observe a linha relativa a ação `show` no comando `rake %routes`:

```
user GET    /users/:id(.:format)      users#show
```

Listando as rotas

No Rails 4, é possível visualizar todas as rotas de sua aplicação dentro do browser, basta visitar http://localhost:3000/rails/info/routes. Essa rota só está disponível no ambiente de desenvolvimento.

Isso significa que, qualquer coisa após a segunda barra e que não contenha ponto será disponibilizado como o parâmetro `:id` e, qualquer coisa depois do ponto, virá como o parâmetro `:format`. Veja alguns exemplos:

- `/users/123` - id: `123`, format: `nil`
- `/users/123.json` - id: `123`, format: `"json"`
- `/users/abcdef.xml` - id: `abcdef`, format: `"xml"`

Vamos usar então o `params[:id]` para buscar o objeto no banco de dados. Altere o arquivo `app/controllers/users_controller.rb`:

```ruby
class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
  end

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
```

```
    if @user.save
      redirect_to @user,
                  notice: 'Cadastro criado com sucesso!'
    else
      render action: :new
    end
  end

  private

  def user_params
    params.
      require(:user).
      permit(:email, :full_name, :location, :password,
             :password_confirmation, :bio)
  end
end
```

E quando o id não existe?

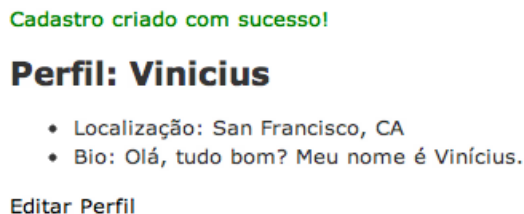
Existe a possibilidade do usuário digitar um id não existente, ou um link estar incorreto. Quando isso acontecer, o `ActiveRecord` irá disparar uma exceção chamada `ActiveRecord::RecordNotFound`. Como já vimos anteriormente, o Rails irá exibir um erro "404 Not Found", que é exatamente o que deve ocorrer quando um objeto não existe.

Vamos ao template. Crie o arquivo `app/views/users/show.html.erb` conforme o código a seguir:

```
<p id="notice"><%= notice %></p>

<h2>Perfil: <%= @user.full_name %></h2>

<ul>
  <li>Localização: <%= @user.location %></li>
  <li>Bio: <%= @user.bio %></li>
</ul>

<%= link_to 'Editar Perfil', edit_user_path(@user) %>
```

A estrutura desse template é um pouco diferente, pois estamos usando menos métodos auxiliares do Rails e mais HTML. Na verdade, a estrutura de templates de aplicações tendem a ser mais dessa forma, uma mistura de HTML com pinceladas de ERB e métodos auxiliares.

O método `notice` é um método auxiliar do Rails para retornar o conteúdo do `flash[:notice]`. Mas o método mais importante para nós neste momento é o método `link_to`.

O método `link_to` serve para gerar links, através da tag <a>. O primeiro atributo é o texto que vai no link, e o segundo é o endereço que o link irá direcionar o usuário.

O método `edit_user_path(@user)` é um método gerado pelo roteador de acordo com seu arquivo de rotas. O método `edit_user_path(@user)` vai gerar a rota correta para a ação de *edit* do usuário `@user`.

Com o template pronto, quando você criar um novo usuário, você verá a seguinte imagem:

Figura 5.5: Perfil do usuário, depois de efetuar um cadastro

5.5 Permita a edição do perfil

A imagem anterior mostra que temos uma tela bem simples, mas já estamos começando a amarrar toda a funcionalidade de cadastro. Ainda faltam as ações de *edit* e *update*. Elas são bem parecidas com as ações *new* e *create*.

Vamos primeiro à ação *edit*. O que precisamos fazer nessa ação é buscar o objeto usuário no banco de dados e então vamos exibir o formulário com os dados que o usuário já preencheu anteriormente. Vamos usar o que já aprendemos nas ações *new* e *show*: para buscar o objeto, vamos usar o mesmo código

da ação *show*. No formulário, basta lembrar que o `form_for` sabe tratar objetos que já existem no banco de dados, populando os campos para que o usuário possa realizar alguma alteração necessária. Vamos então ao `UsersController` (`app/controllers/users_controller.rb`):

```
class UsersController < ApplicationController
  # Omitindo as outras ações para não atrapalhar...

  # show
  # new
  # create

  def edit
    @user = User.find(params[:id])
  end

  # ...
end
```

O template da ação *edit* (`app/views/users/edit.html.erb`) será quase o mesmo da ação *new*, vamos apenas alterar o título:

```
<h1>Editar perfil</h1>

<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %><br />
    <%= f.text_field :full_name %>
  </p>
  <p>
    <%= f.label :location %><br />
    <%= f.text_field :location %>
  </p>
  <p>
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </p>
  <p>
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </p>
```

```
  <p>
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %><br />
    <%= f.text_area :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Agora direcione o browser para a página de edição do modelo. A rota é `/users/:id/edit`, então substitua `:id` pelo ID gerado pelo Rails anteriormente. Teremos o formulário já previamente preenchido com os dados do usuário e, ao clicar no botão *Update User*, que ainda não faz nada.

5.6 Reaproveite as apresentações com partials

Se você comparar o formulário da página *edit* e *new*, verá que as páginas são praticamente idênticas, havendo apenas o título da página de diferença. Não é por acaso, realmente queremos que os dois formulários sejam exatamente iguais.

O problema da solução que temos agora é que, toda vez que quisermos alterar o formulário, será necessário alterar os dois arquivos. Esse processo é um processo altamente propenso a erros, ou seja, é bastante comum que esses dois arquivos não fiquem a par depois de um tempo. Existe uma maneira melhor de resolver esse problema.

O componente de templates do Rails (`ActionView`) possui um recurso chamado *partials*, que são mini-templates que podem ser embutidos em outros templates, ou seja, uma *partial* não é usada para renderizar diretamente uma ação, mas é usada para compor um pequeno pedaço de uma página.

Para criar uma *partial*, basta criar um template com seu nome iniciando em _. Ou seja, nesse caso, vamos criar a *partial* "form". Portanto, crie o arquivo `_form.html.erb`, na pasta `app/views/users/`:

```
<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %><br />
```

```
    <%= f.text_field :full_name %>
  </p>
  <p>
    <%= f.label :location %><br />
    <%= f.text_field :location %>
  </p>
  <p>
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </p>
  <p>
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %><br />
    <%= f.text_area :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Esse formulário é o que já vimos anteriormente. Como a partial está dentro do contexto da ação, a *partial* também possui acesso a todas as variáveis de instância do controle. Como as ações *edit* e *new* usam o mesmo nome de variável, compartilhar a *partial* que acabamos de criar entre as duas ações é bastante simples.

Agora vamos aos templates `app/views/users/new.html.erb` e `app/views/users/edit.html.erb`:

`app/views/users/new.html.erb`:

```
<h1>Cadastro</h1>

<%= render 'form' %>
```

Note que o nome da *partial* não deverá incluir o _. Por convenção, o Rails sabe que nomes de *partials* iniciam com _ e o incluirá automaticamente.

`app/views/users/edit.html.erb`:

```
<h1>Editar perfil</h1>

<%= render 'form' %>
```

Depois de editar os dois arquivos, salve-os e navegue nas páginas. O funcionamento é o mesmo do anterior, porém evitamos o "copiar e colar", que resulta em trabalho repetitivo e propenso a erros.

Agora que os templates estão melhorados, vamos implementar a ação *update*. A ação *update* será bem parecida com a ação *create*, com a única diferença que, ao invés de criarmos um novo objeto, vamos buscar o objeto existente do banco de dados e atualizar seus atributos, através do método `update`. Note que o método `update` também é um método de *mass assignment*, ou seja, atualizará muitos atributos de uma vez. Por essa razão, vamos ter que usar o método `user_params`, que fará a verificação de atributos válidos a serem atualizados.

UPDATE VS. UPDATE_ATTRIBUTES

No Rails 4, foi introduzido o método `update`, que se comporta da mesma maneira que o `update_attributes`. Se você já conhece Rails e está acostumado com a usar `update_attributes`, ele ainda está disponível.

Abra o controle `UsersController` (`app/controllers/users_controller.rb`) e crie o novo método `update`:

```
class UsersController < ApplicationController
  # Omitindo as outras ações para não atrapalhar...

  # new
  # show
  # create

  def edit
    @user = User.find(params[:id])
  end
```

```
  def update
    @user = User.find(params[:id])
    if @user.update(user_params)
      redirect_to @user,
        notice: 'Cadastro atualizado com sucesso!'
    else
      render action: :edit
    end
  end

  # ...
end
```

Se a atualização do perfil ocorrer com sucesso, o usuário é redirecionado para a página do seu perfil, contendo uma mensagem de sucesso, como é possível ver na figura 5.6

Figura 5.6: Atualização de perfil com sucesso

Se na atualização ocorrer algum erro, o formulário de edição será exibido novamente, com marcações em vermelho de forma que o usuário possa corrigir os dados.

5.7 Mostre os erros no formulário

Quando o formulário é enviado com dados que não passam nas validações, os campos ficam marcados com a cor vermelha. Isso acontece pois o `form_for` automaticamente adiciona classes CSS aos campos do formulário que possuem erros. Isso é possível graças ao método `errors`, que vimos anteriormente.

O *scaffold* que geramos no capítulo 3 já cria estilos CSS de forma que os campos

do formulário fiquem com a cor vermelha, destacando para o usuário quais campos necessitam correção.

Figura 5.7: Marcação de erros no formulário

O principal problema desse formulário é que ele não informa exatamente o(s) erro(s) ocorridos, para que o usuário possa tomar uma decisão informada de como corrigir o campo. Sem nenhuma informação, é praticamente impossível que o usuário entenda o que precisa ser alterado. Por isso, vamos adicionar essa dica.

Como a *partial* do formulário é exibida tanto na ação `edit` quanto na `new`, a *partial* criada, `_form`, torna-se uma excelente candidata a receber a exibição de erros. Por isso, vamos adicionar nessa *partial* todas as informações que o método `errors` tiver, caso o objeto esteja inválido. Para isso, abra o arquivo `app/views/users/_form.html.erb` e adicione o seguinte código no início do arquivo:

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    <h2>Infelizmente não foi possível completar
    a ação pois o formulário possui os seguintes erros:</h2>
    <ul>
      <% @user.errors.full_messages.each do |message|%>
```

```
        <li><%= message %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<%= form_for @user do |f| %>
   ...
<% end %>
```

O que esse código faz é verificar se há algum erro (o método `#any?` retorna `true` se pelo menos um elemento de uma `Array` é diferente de `nil`). Se houver um ou mais erros, cria-se a `div` com uma mensagem e em seguida lista todas as mensagens de erro. O resultado é o seguinte:

Figura 5.8: Mensagem de erro com mensagem amigável

Agora o usuário pode ler a mensagem e tomar uma ação para corrigir seus erros. O funcionamento das páginas está correto, porém vamos adicionar links para melhorar a navegação.

5.8 Configure a ação raiz (root)

A ação raiz, ou *root*, é a ação que é executada quando vamos ao endereço raiz do site, por exemplo, http://www.colcho.net, ou http://localhost:3000/. O que temos agora é ainda a página de boas vindas do Rails. Vamos mudar isso.

Por padrão, o Rails define a rota raiz como a página que vimos. Basta criar a entrada `root` no arquivo `config/routes.rb` para apontar para a página principal, que ainda vamos criar:

```
Colchonet::Application.routes.draw do
  resources :rooms
  resources :users

  root 'home#index'
end
```

A notação de controles, quando mencionados manualmente, é o seu nome (por exemplo, "users" para o `UsersController`), sustenido (#, ou também 'jogo da velha') e o nome da ação. Dessa forma, "home#index" aponta para o controle `HomeController`, ação *index*. Crie o arquivo `app/controllers/home_controller.rb`:

```
class HomeController < ApplicationController
  def index
  end
end
```

Boa prática: sempre declare todas as ações

Quando uma ação não faz nada, é possível simplesmente não implementar nenhum método no controle e apenas criar o template. O `HomeController`, por exemplo, não necessitaria do método `index`, pois tem o template para exibir a página e não contém absolutamente nada.

Porém, é sempre interessante declarar todos os métodos que o controle responde, para que fique claro ao leitor do código quais ações aquele controle responde.

Agora, crie a pasta `app/views/home` e o template `app/views/home/index.html.erb`:

```
<h1>Colcho.net</h1>

<p>
  Escolha uma das ações a seguir:
  <ul>
    <li><%= link_to "Cadastro", new_user_path %></li>
    <li><%= link_to "Visualizar quartos", rooms_path %></li>
  </ul>
</p>
```

Agora temos links para as principais páginas da nossa aplicação até o momento. Vamos adicionar dois novos links, para que o usuário possa desistir de cadastrar ou atualizar o seu perfil:

`app/views/users/new.html.erb`:

```
<h1>Cadastro</h1>

<%= render 'form' %>

<%= link_to "Voltar", root_path %>
```

`app/views/users/edit.html.erb`:

```
<h1>Cadastro</h1>

<%= render 'form' %>

<%= link_to "Voltar", @user %>
```

Note que, como as rotas são diferentes, não colocamos na *partial*, mas no template em si. Na ação `edit` usamos a capacidade do Rails de determinar a rota de `show` de um recurso, apenas colocando o objeto como parâmetro.

Por fim, vamos editar a página do perfil do usuário, `app/views/users/show.html.erb`, adicionando um link ao final:

```
...

<%= link_to 'Editar Perfil', edit_user_path(@user) %>
<%= link_to 'Home', root_path %>
```

A rota `root_path` é especial, e aponta para a raiz do site. Pronto, agora temos interligação entre todas as páginas do site, inclusive à área de quartos que deixamos intocado por enquanto. Agora vamos focar apenas nos modelos, aprimorando a qualidade dos dados armazenados.

Capítulo 6

Melhore o projeto

Se longe enxerguei foi porque me apoiei no ombro de gigantes.
– Isaac Newton

6.1 Lição obrigatória: sempre aplique criptografia para armazenar senhas

Eu tenho calafrios toda vez que faço um cadastro em um site e recebo um email com a minha senha, que deveria ser algo secreto. O problema que este email revela é grave: a senha está sendo armazenada em texto puro, sem nenhuma cifra ou encriptação. Mesmo sendo um problema resolvido, não é rara a notícia de que algum site de alguma importância tenha seu banco de dados vazado e distribuídos. E o colcho.net, por enquanto, não é nem um pouco diferente.

```
User.first
# <User id       : 1,
#   full_name    : "Vinicius",
```

```
#    email      : "vinibaggio@example.com",
#    password   : "segredo",
#    location   : "San Francisco, CA",
#    bio        : "O meu perfil anterior...",
#    created_at : "2012-06-21 05:37:34",
#    updated_at : "2012-06-22 06 : 36 : 59">
```

Para resolver esse problema, basta usar uma funcionalidade embutida no Rails, que usa a encriptação BCrypt. Essa funcionalidade chama-se `has_secure_password` e, para usá-la, precisamos fazer as seguintes alterações no modelo Usuário:

1) Usar a funcionalidade `has_secure_password` no modelo;

2) Criar a coluna `password_digest`;

3) Cifrar a senha atual do usuário;

4) Deletar a coluna `password`;

Encriptação de senha

O algoritmo usado pelo Rails para cifrar senhas é o BCrypt. Nesse algoritmo, a senha em texto puro passa por cálculos matemáticos que a torna cifrada. Esse processo é irreversível, ou seja, uma vez calculada a cifra, é impossível fazer a volta.

Para verificar a validade de uma senha, fornecemos o que o usuário digitar ao algoritmo. Comparamos o resultado final com o que está armazenado. Se os resultados forem os mesmos, o usuário digitou a senha corretamente.

O BCrypt é usado pois as chances de colisão são muito pequenas, ou seja, duas cadeias de caracteres diferentes raramente terão a mesma cifra final. Por isso, se o banco de dados vazar, não será possível usar as senhas dos usuários.

Este não é um processo infalível mas é estatisticamente seguro. Antigamente usava-se algoritmos como MD5 e SHA1, mas já provou-se que são ruins, pois há maiores chances de colisão.

Por fim, **jamais** invente o seu próprio mecanismo, por mais que a ideia soe tentadora. Criptografia é um dos problemas mais difíceis da computação e eu recomendo que deixamos isso na mão de especialistas.

6.2 Como adicionar plugins ao projeto?

O Rails não implementa o algoritmo BCrypt, portanto é necessário instalar uma gem que implementa esse algoritmo. A gem é a `bcrypt-ruby` e, para que o Rails consiga usá-la, abra o arquivo `Gemfile`:

```
# To use ActiveModel has_secure_password
gem 'bcrypt-ruby', '~> 3.0.0'
```

Em seguida, no terminal, vá à raiz do projeto e execute o comando `bundle`:

```
$ bundle
Using rake (10.1.0)
Using i18n (0.6.4)
Using minitest (4.7.5)
```

```
Using multi_json (1.7.7)
Using atomic (1.1.10)
...
Using arel (4.0.0)
Using activerecord (4.0.0)
Installing bcrypt-ruby (3.0.1) with native extensions
Using bundler (1.3.0)
...
Your bundle is complete!
```

Esse comando instalará a gem em seu sistema. O arquivo `Gemfile` e o comando `bundle` fazem parte de uma ferramenta chamada Bundler (http://gembundler.com/) . Ela é uma ferramenta para o gerenciamento de dependências de aplicativos Ruby. O Rails usa-o para carregar todas as bibliotecas com as versões corretas. O trabalho pesado fica para a ferramenta, nos resta apenas manter o arquivo `Gemfile` organizado.

Após a execução do `bundle`, o bundler irá gerar uma nova versão do `Gemfile.lock`, que contém exatamente todas as dependências e versões de gems que satisfazem as dependências da aplicação Rails. Abrindo esse arquivo, dá para ver todas as dependências de uma gem.

Reiniciando o console e o servidor

Em mudanças como essa, é necessário reiniciar tanto o console quanto o servidor, mesmo com o uso do `reload!`. Como o Rails carrega suas dependências no momento que é iniciado, portanto o `reload!` conseguirá levar em conta essas alterações.

6.3 Usando has_secure_password no modelo

Instalada a gem, alteraremos o modelo. Segundo a documentação do `has_secure_password` (procure por `has_secure_password` no site http://api.rubyonrails.org para detalhes), essa *class macro* já nos dá as validações de confirmação de senha e a presença de senha. Além disso, ela cria dois novos atributos virtuais, o `password` e `password_confirmation`.

Anteriormente fazíamos essas validações manualmente, portanto temos que tirá-las, substituindo pelo uso da *class macro* `has_secure_password`. Como im-

plementamos usando os mesmos nomes de atributos, não é necessário alterar nenhum formulário.

Alteremos o modelo usuário (`app/model/user.rb`) da seguinte forma:

```
class User < ActiveRecord::Base
  EMAIL_REGEXP = /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/

  # Lembre-se de tirar as validações de senha!
  # O has_secure_password já o faz para você.
  validates_presence_of :email, :full_name, :location
  validates_length_of :bio, minimum: 30, allow_blank: false
  validates_format_of :email, with: EMAIL_REGEXP

  has_secure_password
end
```

Com essas alterações no modelo, já temos todo o código necessário para encriptar as senhas, só resta as alterações de banco de dados.

6.4 Migração da tabela users

O `has_secure_password` requer que modelos possuam a coluna `password_digest` no banco de dados, portanto vamos criá-la. Execute, na pasta raiz do projeto, o comando que irá gerar a migração:

```
$ rails g migration add_password_digest_to_users password_digest
   invoke  active_record
   create    db/migrate/20130719050850_add_password_digest_to_users.rb
```

A migração irá criar um arquivo. O nome do dele irá variar conforme a hora que o comando foi executado, mas a saída irá mostrar o caminho que você pode usar para editá-lo.

```
class AddPasswordDigestToUsers < ActiveRecord::Migration
  def change
    add_column :users, :password_digest, :string
  end
end
```

Executemos a migração com a tarefa rake `rake db:migrate`:

```
$ rake db:migrate
==  AddPasswordDigestToUsers: migrating =================
-- add_column(:users, :password_digest, :string)
   -> 0.0292s
==  AddPasswordDigestToUsers: migrated (0.0296s) ========
```

Por que não fizemos certo desde o início?

Essa é uma pergunta natural de ser feita, e a resposta é simples. É raro o dia em que sabemos toda a modelagem do nosso sistema logo de início. É certo que um dia será necessário alterar o modelo por causa de alguma decisão tomada ou algum conhecimento adquirido. O Rails dá ferramentas para dar suporte a esse tipo de desenvolvimento, ao invés de gastar dias e dias com modelagem, o Rails abraça a ideia do crescimento orgânico de um sistema.

Uma vez o banco de dados migrado, podemos usar a funcionalidade por completo. No exemplo a seguir, vamos criar um novo objeto usando os métodos `password=` e `password_confirmation=`. Em seguida, vamos buscar esse objeto do banco de dados e verificar se a senha está correta. Vamos usar o método `authenticate`, adicionado automaticamente pela *class macro* `has_secure_password`, que irá retornar false quando a senha estiver inválida e o próprio objeto quando estiver correta:

```
v = User.new
# => #<User id: nil,
#           full_name: nil,
#           email: nil,
#           location: nil,
#           bio: nil,
#           created_at: nil,
#           updated_at: nil,
#           password_digest: nil,
#           password: nil>

v.full_name = 'Vinicius Baggio Fuentes'
# => "Vinicius Baggio Fuentes"
v.email = 'vinibaggio@example.com'
```

```
# => "vinibaggio@example.com"
v.location = 'San Francisco, CA, EUA'
# => "San Francisco, CA, EUA"
v.bio = 'Desenvolvedor Web, principalmente com Ruby on Rails'
# => "Desenvolvedor Web, principalmente com Ruby on Rails"
v.password = 'segredo'
# => "segredo"
v.password_confirmation = 'segredo'
# => "segredo"
v.save!

u = User.find_by_email 'vinibaggio@example.com'
# => #<User id: 9, ... >

u.authenticate 'invalido'
# => false

u.authenticate 'segredo'
# => #<User id: 9, ... >
```

Vamos usar esse mecanismo no capítulo 9 com mais detalhes para fazer o controle do login do usuário, mas já podemos garantir a qualidade do armazenamento das senhas. Em seguida, vamos atualizar os dados antigos, de modo a funcionarem com o novo mecanismo.

6.5 Automatizando tarefas de manutenção com rake

O sistema neste momento ainda possui as senhas antigas armazenadas no campo `password`. Porém, agora que usamos o `has_secure_password`, o método `password` já não age da forma anterior -- retornar o resultado da coluna no banco de dados:

```
user
# => #<User id: 8,
#   full_name: "Teste",
#   email: "vinibaggio+123@example.com",
#   password: "123123",
#   location: "teste",
#   bio: "Este é um teste de bio para o Colcho.net",
```

```
#   created_at: "2012-12-12 06:22:44",
#   updated_at: "2012-12-12 06:22:44",
#   password_digest: nil>

user.password
# => nil
```

Por isso, para acessar os dados brutos do banco de dados, podemos usar o método `attributes`, que retorna a hash com todos os valores:

```
user.attributes
{"id"=>8,
 "full_name"=>"Teste",
 "email"=>"vinibaggio+123@gmail.com",
 "password"=>"123123",
 "location"=>"teste",
 "bio"=>"Este é um teste de bio para o Colcho.net",
 "created_at"=>Wed, 12 Dec 2012 06:22:44 UTC +00:00,
 "updated_at"=>Tue, 22 Jan 2013 05:49:33 UTC +00:00,
 "password_digest"=>nil}
```

Para encriptar a senha que temos no banco de dados, vamos criar uma tarefa `rake`. As tarefas `rake` são comandos executados no terminal que alteram o estado atual do sistema. Todas as tarefas `rakes` devem ficar na pasta `lib/tasks` e devem ser um arquivo do tipo `.rake`.

A tarefa `rake` precisará ler a senha do banco de dados e vamos associar aos atributos `password` e `password_confirmation` de todos os usuários. Isso fará com que o `has_secure_password` faça o trabalho de encriptar a senha. Criaremos o arquivo `lib/tasks/migrar_senhas.rake`:

```
# encoding: utf-8
namespace :app do
  desc "Encripta todas as senhas \
        que ainda não foram processadas \
        no banco de dados"
  task migrar_senhas: :environment do

  end
end
```

Observação: não esqueça do comentário na primeira linha!

O comentário na primeira linha do arquivo não é opcional. O comentário `# encoding: utf-8` indica ao interpretador Ruby que o arquivo deverá ser processado como `utf-8` e, portanto, aceitará caracteres não ASCII. Lembre-se de configurar seu editor para salvar o arquivo em modo `utf-8`!

A anatomia de tarefas `rake` é a seguinte:

- Um `namespace`, ou seja, um nome para agrupar todas as suas tarefas, de modo que não haja conflito com tarefas previamente definidas. Nesse caso, chamamos de `app`.
- Uma descrição da tarefa, para tentar explicar de forma sucinta o que a tarefa irá fazer, via o comando `desc`.
- A tarefa deve vir logo em seguida à chamada `desc`. O parâmetro é o nome da tarefa que vamos executar no console, junto com a sua dependência. Nesse exemplo, o nome da tarefa é `migrar_senhas` e ela depende da tarefa `environment`, que carregará o ambiente Rails.

Com apenas isso já é possível observar o resultado quando executamos a listagem de tarefas `rake` do projeto:

```
$ rake -T
...
rake app:migrar_senhas      # Encripta todas as
                              senhas que ainda não
                              foram processadas no
                              banco de dados
...
```

Vamos ao código. Primeiro, vamos verificar se as senhas já foram migradas. Vamos usar a existência da coluna `password` no banco de dados. Se ela não mais existir, todas as migrações já foram executadas e portanto não é possível mais migrar os dados.

Em seguida, vamos alterar todos os usuários no banco de dados. Usaremos o método `find_each`, que busca objetos em lotes de 1000. Evitamos assim carregar todos os objetos em memória, que pode causar problemas sérios em um servidor.

Com o objeto em mãos, basta ler o atributo do banco de dados diretamente, sem usar os acessores, e sobrescrevemos, usando os métodos gerados pelo `has_secure_password`. O resultado é o seguinte:

```
# encoding: utf-8
namespace :app do
  desc "Encripta todas as senhas que
       ainda não foram processadas no banco de dados"
  task migrar_senhas: :environment do
    unless User.attribute_names.include? "password"
      puts "As senhas já foram migradas, terminando."
      return
    end

    User.find_each do |user|
      puts "Migrando usuário ##{user.id} #{user.full_name}"
      unencripted_password = user.attributes["password"]

      user.password = unencripted_password
      user.password_confirmation = unencripted_password
      user.save!
    end
  end
end
```

Para executar, vá ao terminal e execute:

```
$ rake app:migrar_senhas
Migrando usuário #6 João
Migrando usuário #7 Teste
Migrando usuário #8 Teste
```

Erros de validação

É muito provável que você tenha erros de validação e a tarefa `rake` termine abruptamente. Isso é esperado, já que é possível que você tenha criado usuários antes de termos criados as validações. Você tem duas opções: pode corrigir os modelos que estão gravados no banco ou apenas pode deletá-los.

Uma vez os dados migrados, podemos criar a migração para remover a coluna que não usamos mais. O procedimento é o mesmo do que fizemos anteriormente -- geraremos uma migração, adicionaremos o código e a executaremos:

```
$ rails g migration remove_password_from_users
  invoke  active_record
  create    db/migrate/20130719051308_remove_password_from_users.rb
```

```
class RemovePasswordFromUsers < ActiveRecord::Migration
  def up
    remove_column :users, :password
  end

  def down
    add_column :users, :password, :string
  end
end
```

Observe que nessa migração não usamos o método `#change`. O Rails não consegue gerar a migração contrária a partir da remoção da coluna, já que não especificamos o tipo da coluna. Por esse motivo, criamos os métodos `#up` e `#down` manualmente.

```
$ rake db:migrate
==  RemovePasswordFromUsers: migrating ================
-- remove_column(:users, :password)
   -> 0.0453s
==  RemovePasswordFromUsers: migrated (0.0454s) =======
```

6.6 Melhoria de templates e CSS

Até agora estamos usando o CSS gerado pelo *scaffold* do Rails. Embora seja funcional, ele é bem cru:

Figura 6.1: Site com o layout do scaffold

Os formulários são visualmente agressivos:

Figura 6.2: Formulário com erro

Com um pouco de HTML e CSS, deixaremos o site mais bonito (considerando a minha falta de habilidade com artes visuais):

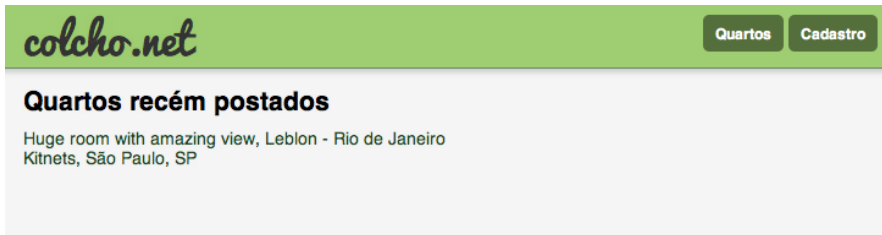



Figura 6.3: Nova home

Vamos também melhorar a exibição de erros para que as mensagens fiquem lado a lado aos campos a serem corrigidos:

Figura 6.4: Formulário com melhor notificação de erros

Para fazer isso, trabalharemos na camada de apresentação do site, alterando templates, criando CSS e criando métodos auxiliares para tornar a construção do template mais fácil.

Conhecimentos de HTML e CSS

Infelizmente não dá para explicar em detalhes cada parte do que vamos fazer em seguida, pois este não é o foco do livro. Se você gostaria de aprender mais sobre esses assuntos (recomendado se você quiser se tornar um profissional completo de web), recomendo a leitura do livro *HTML5 e CSS 3: Domine a web do futuro*, escrito pelo Lucas Mazza.

6.7 Trabalhe com layout e templates para melhorar sua apresentação

O primeiro passo é alterar o layout da aplicação para incluir o cabeçalho com o menu, na direita. Além disso, vamos:

- Incluir o código CSS usar a fonte "Pacifico", disponível no Google Web Fonts, para o logotipo (não se preocupe se você não tiver acesso a internet, vamos usar fontes *fallback*);
- A criação da tag `header` e seu conteúdo, o logotipo e o menu de navegação;
- Embrulhar o `yield` em uma tag `content`, para centralizar todo o conteúdo dos templates.

Altere o arquivo `app/views/layouts/application.html.erb` para que ele fique da seguinte forma:

```
<!DOCTYPE html>
<html>
<head>
  <title>ColchoNet</title>
  <link href='http://fonts.googleapis.com/css?family=Pacifico'
                    rel='stylesheet' type='text/css'>
  <%= stylesheet_link_tag "application",
                         media: "all",
                         "data-turbolinks-track" => true %>

  <%= javascript_include_tag "application",
                            "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<body>
  <header>
    <div id="header-wrap">
      <h1><%= link_to "colcho.net", root_path %></h1>
      <nav>
        <ul>
          <li><%= link_to "Quartos", rooms_path %></li>
          <li><%= link_to "Cadastro", new_user_path %></li>
        </ul>
```

```
      </nav>
    </div>
  </header>

  <div id="content">
    <% if notice.present? %>
      <p id="notice"><%= notice %></p>
    <% end %>
    <% if alert.present? %>
      <p id="alert"><%= alert %></p>
    <% end %>

    <%= yield %>
  </div>
</body>
</html>
```

Vamos tirar a referência ao `flash` da ação `show` do usuário controle de usuários e alterar a tag `h2` para `h1`:

```
<h1>Perfil: <%= @user.full_name %></h1>

<ul>
  <li>Localização: <%= @user.location %></li>
  <li>Bio: <%= @user.bio %></li>
</ul>

<%= link_to 'Editar Perfil', edit_user_path(@user) %>
```

Como agora temos o nome do site e o menu em todas as páginas, aquela página inicial (ou "home") que criamos na seção 5.2 fica redundante. Vamos alterá-la para mostrar ao usuário até três quartos que já foram cadastrados.

Antes de alterar o template, porém, precisamos alterar o controle `HomeController` para buscar os três quartos mencionados anteriormente. Usaremos o método `.take` que retornará objetos do banco de dados, sem ordem determinada. Podemos passar um número para esse método, que irá limitar o número de objetos retornados. Por fim, a consulta SQL resultante é:

```
SELECT "rooms".* FROM "rooms" LIMIT 3
```

Edite portanto o arquivo `app/controllers/home_controller.rb`:

```
class HomeController < ApplicationController
  def index
    @rooms = Room.take(3)
  end
end
```

Agora vamos editar o template do controle `HomeController`, ação *index* (arquivo `app/views/home/index.html.erb`):

```
<h1>Quartos recém postados</h1>
<ul>
  <% @rooms.each do |room| %>
    <li><%= link_to "#{room.title}, #{room.location}", room %></li>
  <% end %>
</ul>
```

Com essas alterações será possível observar a listagem de quartos, caso você tenha cadastrado algum objeto no banco de dados. Porém, há um pequeno problema no template anterior que devemos corrigir.

Você pode estar se perguntando: "qual o problema desse código?". E a resposta é simples: esse é um típico caso em que estamos violando uma regra de negócio que parece inofensiva: o nome completo de um quarto. Mas em casos como esse é comum que os programadores acabem repetindo a composição diversas vezes em templates espalhados. E o dia que a lógica for alterada, será necessário realizar vários "buscar e substituir". Para manter o código consistente, vamos fazer do jeito certo: o template deve saber o **mínimo** possível de como o modelo é feito.

Abra agora o então intocado modelo `Room` (`app/models/room.rb`) e adicione o método `#complete_name`:

```
class Room < ActiveRecord::Base
  def complete_name
    "#{title}, #{location}"
  end
end
```

Voltando ao template (`app/views/home/index.html.erb`):

```
<h1>Quartos recém postados</h1>
<ul>
  <% @rooms.each do |room| %>
    <li><%= link_to room.complete_name, room %></li>
```

```
  <% end %>
</ul>
```

Antes de começarmos a modificar os *stylesheets* (CSSs) da aplicação, vamos parar um pouco e entender mais sobre um dos componentes mais controversos, em minha opinião, do Rails.

6.8 O que é o Asset Pipeline?

O *Asset Pipeline* veio para resolver um problema complicado da web moderna: a entrega de *assets*, ou arquivos estáticos. O que parece simples como a entrega de um arquivo javascript ou stylesheet ao browser do usuário pode se tornar uma tarefa complicada.

Como as aplicações web tem se tornado complexas, diversas ferramentas estão sendo usadas para tornar o desenvolvimento de javascripts e *stylesheets* mais produtivo. Além disso, é necessário diminuir ao máximo o tempo que o browser leva para baixar e processar esses arquivos, para que as aplicações web não passem a percepção de serem lentas.

Dessa forma, é comum que aplicações web possuam uma ou mais das seguintes fases para a entrega de um arquivo estático:

1) Pré-compilação: Transformação de um SCSS ou LESS em CSS puro, ou CoffeeScript em JavaScript;

2) Concatenação: Juntar todos os arquivos CSS ou JavaScript em apenas um;

3) "Minificação": uso de técnicas para diminuir o tamanho dos scripts, como renomear variáveis (de "essaVariavelEhGrande" para "a");

4) Compressão: Usar compressão GZip para diminuir ainda mais o tamanho do arquivo, caso o browser tenha suporte (todos os browsers modernos possuem esse suporte).

SCSS e CoffeeScript são opcionais

Por padrão, o Rails já instala SASS (e por consequência, SCSS) e CoffeeScript em seu projeto. Porém, o uso de SCSS e CoffeeScript é opcional, ou seja, você pode continuar escrevendo JavaScript e CSS puro e ainda aproveitar dos outros benefícios do Assets Pipeline.

Em conjunto com o processo anterior, existe outra preocupação: *caching*. Além do seu browser, existem diversos elementos de rede que podem fazer *caching* de *assets*, como seu provedor de internet, *firewalls*, CDNs (*Content Delivery Network*, ou rede de entrega de conteúdo), equipamentos de rede (roteadores), entre outros. O problema é que, quando você atualiza um arquivo no servidor, o conteúdo desse arquivo pode estar em *cache* e portanto os usuários do site podem receber um conteúdo antigo.

O *Asset Pipeline* ajuda na resolução de todos os problemas mencionados, desde que estejamos dispostos a cooperar com ele, ou seja, seguir as suas convenções.

Primeiro, o Rails, através de uma biblioteca chamada `sprockets`, lê cada manifesto da sua aplicação; um arquivo que contém diretivas declarando quais são os arquivos em que seu aplicativo depende.

Com essa lista de arquivos, o Rails compila cada um de acordo com as extensões utilizadas. Por exemplo, se o nome do seu *stylesheet* for `users.css.scss.erb`, o *stylesheet* vai primeiro ser pré-processado em ERb e, em seguida, SASS (SASS e SCSS são pré-processados pelo próprio SASS), gerando um arquivo CSS final.

Após obter todos os arquivos pré-compilados, o Rails irá juntar todos os arquivos de um manifesto e gerará um arquivo com mesmo nome. Por exemplo, se o manifesto `application.css` mencionar os arquivos `footer.css` e `reset.css` e o manifesto `admin.css` mencionar apenas o `reset.css`, o resultado final desse processo serão dois arquivos, o `application.css`, contendo o `footer.css` e o `reset.css` concatenados e o `admin.css`, apenas o `reset.css`.

A razão desse processo é que cada browser possui um limite de quantos arquivos ele baixará simultaneamente. Por exemplo, se o seu site usa dez imagens e o browser possui limite de cinco downloads simultâneos, o sexto arquivo só será baixado a partir do momento que um dos cinco downloads anteriores terminarem. Esses valores podem variar de acordo com o browser e com o seu site, mas em geral, concatenar os arquivos dessa forma é uma boa estratégia.

Após a concatenação, o que acontece é a minificação. O Rails (ou mais especificamente, o `sprockets`) gera uma versão minificada de cada arquivo gerado no passo anterior. O processo de minificação gera um arquivo javascript ou *stylesheet* completamente válido, mas ilegível. Reduzir o arquivo dessa maneira possui duas implicações importantes: diminuir o tempo que o browser tem para baixar o arquivo e, consequentemente, inicia o download de outros arquivos bloqueados pelo browser, se houver.

Chegando ao final do processo, o Rails calcula o *digest* MD5 do arquivo e adici-

ona ao nome do arquivo. Por exemplo, se o arquivo chama-se `application.css`, ele se tornará algo parecido com:

```
application-e049a640704156e412f6ee79daabc7f6.css
```

Esse código gerado depende do conteúdo do arquivo, portanto se uma nova versão desse *asset* for gerada, o código será alterado. Isso fará com que o mecanismo de *caching* não seja acionado caso uma nova versão do *asset* seja compilada.

Por fim, há a compressão desses *assets* com o algoritmo GZip, gerando:

```
application-e049a640704156e412f6ee79daabc7f6.css.gz
```

Com o suporte a GZip de servidores como Apache (http://httpd.apache.org/) ou nginx (pronuncia-se "engine x", ou ênginéx) (http://http://nginx.com/) e dos browsers, a entrega desses *arquivos* torna-se muito mais eficiente, diminuindo ainda mais o tempo total de carga.

Note que, para esse mecanismo funcionar, é estritamente necessário que executemos a pré-compilação desses *assets* nos servidores:

```
rake assets:precompile
```

Isso irá gerar os arquivos de *assets*, que não serão entregues pelo Rails, mas sim pelo seu servidor web.

No modo de desenvolvimento de aplicações Rails, que é o modo que estamos executando, as coisas são um pouco mais simples. Ao invés de serem concatenados e compactados, os *assets* são apenas pré-compilados automaticamente por cada página. É possível ver no HTML gerado que há tags de inclusão de javascript/stylesheets para cada arquivo mencionado no manifesto.

Agora que entendemos um pouco do que é o Asset Pipeline, vamos usá-lo para criar *stylesheets* para o Colcho.net.

6.9 Criando os novos stylesheets

Abra o arquivo `app/assets/stylesheets/application.css`. Você pode ver que não há nenhum código CSS de fato, mas há um conjunto de comentários importantes que estão lá para compor o que chamamos de "manifesto":

```
/*
 *= require_self
 *= require_tree .
 */
```

Esses dois comentários são diretivas do `sprockets` (note o "=" na linha). O `require_self` faz com que qualquer CSS que esteja no arquivo do manifesto seja incluído antes do restante, e o `require_tree` varre toda a árvore de diretórios a partir do diretório especificado (no caso, ".", ou pasta atual) e inclui os *stylesheets* no manifesto.

Isso nos diz, em termos práticos, que qualquer CSS que incluirmos na pasta `app/assets/stylesheets/` será incluído no manifesto final, e isso é suficiente para nós.

Não vamos utilizar o estilo gerado pelo *scaffold*, então vamos deletá-lo. Para isso, basta apagar o arquivo `app/assets/stylesheets/scaffolds.css.scss`. Você pode apagar também o CSS gerado pelo *scaffold* para o controle `RoomsController`: `app/assets/stylesheets/rooms.css.scss`.

Vamos começar a criar o nosso próprio *stylesheet*. Como a maioria dos browsers não possuem padrões razoáveis, é comum usar um *stylesheet* chamado *reset*, que deixa todos os estilos em um mesmo padrão para que, a partir daí, possamos construir o nosso próprio.

É recomendado usar um *reset* popular e que já foi testado em vários browsers. O "Eric Meyer's Reset" (http://meyerweb.com/eric/tools/css/reset/) é recomendado, porém, caso você não possua internet, vamos usar um *reset* bem simplificado. Crie o arquivo `app/assets/stylesheets/reset.css` da seguinte forma:

```
* {
  margin: 0;
  padding: 0;
  text-decoration: none;
}
li { list-style: none; }
```

Esse CSS deixa o `padding` e a `margin` em valores zerados. Não contempla diversos browsers, porém serve como uma base. Em seguida, veremos em blocos o CSS final do site.

Crie o arquivo `app/assets/stylesheets/default.css.scss`. Note que esse não é um arquivo de CSS comum e portando **deve** ter a extensão `.scss`. Isso significa que esse CSS é um SCSS, uma extensão de CSS que adiciona funcionalidades como regras aninhadas, variáveis, *mixins* e outros. Para entender como funciona o SCSS, veja o site oficial do projeto SASS (http://sass-lang.com) .

A primeira parte é a declaração de variáveis em SCSS, de forma a centralizar algumas configurações globais:

```
$header-height: 55px;
$content-width: 700px;

$serif-families: "Pacifico", "Georgia", serif;
$sans-serif-families: "Helvetica", sans-serif;

$error-text-color: #B94A48;
$success-text-color: #468847;
```

Em seguida, declaramos o estilo geral da página, como cor do fundo, tamanho e família da fonte do texto do site (usando a variável `$sans-serif-families`):

```
* {
  font-family: $sans-serif-families;
  font-size: 14px;
}

body { background-color: #f5f5f5; }
```

Declaramos um *mixin* para facilitar a declaração de `box-shadow`, colocando todos os *vendor prefixes* quando necessário:

```
@mixin shadow($color, $x, $y, $radius) {
  -moz-box-shadow:    $color $x $y $radius;
  -webkit-box-shadow: $color $x $y $radius;
  box-shadow:         $color $x $y $radius;
}
```

Em seguida, vamos criar a barra de navegação, que fica no topo. Criamos a cor de fundo da barra e uma sombra na parte inferior. Criamos também o `#header_wrap`, que é responsável pelo alinhamento do conteúdo da barra de navegação em conjunto com o conteúdo. Por fim, estilizamos o logotipo. Note que vamos usar o *mixin* criado anteriormente via `@include`, evitando ter que repetir três regras para cada *vendor prefix*.

```
header {
  @include shadow(#ccc, 0, 3px, 6px);

  border-bottom: 1px solid #686;
  margin-bottom: 15px;
```

```
  #header-wrap {
    width: $content-width;
    margin: 0 auto;
  }

  height: $header-height;
  background-color: #9ECE71;
  h1 {
    float: left;
    a {
      color: #333;
      font-family: $serif-families;
      font-weight: 400;
      font-size: 2.5em;
      &:hover {
        color: #000;
      }
    }
  }
}
```

A segunda parte da barra superior, o estilo do menu de navegação:

```
header nav {
  float: right;

  ul {
    display: inline-block;
    vertical-align: middle;
    line-height: 55px;
  }

  li {
    display: inline-block;
    line-height: 24px;
    background-color: #546f3c;
    padding: 3px 10px;

    -moz-border-radius: 5px;
    border-radius: 5px;
    vertical-align: middle;
```

```
    a {
      display: block;
      font-size: 12px;
      font-weight: 600;
      color: #fff;
    }
  }
}
```

O estilo do conteúdo das páginas ficará dentro da `div` com id `content`, como deixamos no layout da aplicação (`app/views/layouts/application.html.erb`). Note como podemos fazer referência a um seletor pai usando `&`.

```
#content {
  text-align: left;
  width: $content-width;
  margin: 0 auto;

  h1 {
    font-size: 1.5em;
  }

  a, a:visited, a:hover {
    color: #242;
    &:hover {
      text-decoration: underline;
    }
  }

  ul, form, p {
    margin: 10px 0;
  }
}
```

Para o formulário, temos o seguinte CSS:

```
form {
  label {
    display: block;
    margin: 5px 0;
```

```
    color: #444;
  }

  input[type=text], input[type=password], textarea {
    color: #444;
    font-size: 12px;
    border: 1px solid #ccc;
    padding: 5px;
    width: 200px;
    outline: 0;

    @include shadow(rgba(0,0,0, 0.1), 0px, 0px, 8px);
    &:focus {
      border: 1px solid #c9c9c9;
    }
  }

  textarea {
    width: 400px;
    height: 200px;
  }
}
```

Por fim, temos as classes relacionadas com a informação de erros e `flash`:

```
.field_with_errors {
  display: inline;

  label { color: $error-text-color; }

  input[type=text], input[type=password], textarea {
    border: 1px solid rgba(189,74,72, 0.5);
    @include shadow(rgba(189,74,72, 0.2), 0px, 0px, 8px);

    &:focus { border: 1px solid rgba(189,74,72, 0.6); }
  }
}

.error_message {
  margin-left: 5px;
  display: inline;
  color: $error-text-color;
```

```
}

.padded_flash {
  padding: 10px;
  margin: 10px 0;
  font-weight: bold;
  width: 500px;
}

#error_explanation {
  border: 1px solid $error-text-color;
  color: $error-text-color;
  background-color: #F2DEDE;
  @extend .padded_flash;
}

#notice {
  color: $success-text-color;
  border: 1px solid $success-text-color;
  @extend .padded_flash;
  background-color: #DFF0D8;
}

#alert {
  color: $error-text-color;
  border: 1px solid $error-text-color;
  @extend .padded_flash;
  background-color: #F2DEDE;
}
```

Salve o arquivo e navegue. Agora sim, bem melhor! Mas ainda não concluímos, precisamos corrigir o formulário de erros, que exigirá algumas mudanças no template.

6.10 Feedback em erros de formulário

Vamos limpar o formulário de cadastro de usuário. Removeremos as tags `<br/>`, pois os *labels* já possuem layout de bloco (`display: block`) e simplificaremos a notificação de erros, já que fica difícil para o usuário entender o que deverá corrigir. Colocando as mensagens logo ao lado do campo, o usuário consegue agir muito mais

rapidamente. Veja o exemplo para o campo "*full name*", ou "nome completo":

```
<p>
  <%= f.label :full_name %>
  <%= f.text_field :full_name %>
  <% if @user.errors.has_key? :full_name %>
    <div class="error_message">
      <%= @user.errors[:full_name].first %>
    </div>
  <% end %>
</p>
```

No código anterior, verificamos se há algum erro no atributo `full_name`. Caso isso seja verdade, criamos uma tag `div` com classe `error_message` e mostramos então o primeiro erro, apenas para simplificar a tela.

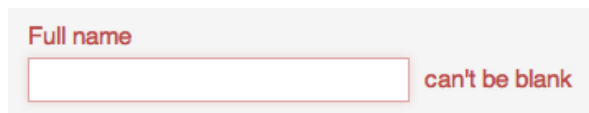



Figura 6.5: Campo com notificação de erro

6.11 Duplicação de lógica na apresentação nunca mais. Use os Helpers

Ao expandir as alterações para exibição de erros para os outros campos, nota-se como o código final fica repetitivo. Os problema é que, para cada campo do formulário, repetimos uma lógica (um `if`). Fazer uma alteração parecida para o formulário de quartos seria bastante trabalhoso e fica clara a necessidade de se corrigir este problema.

Não faz sentido, porém, colocar esse tipo de lógica em qualquer uma das camadas que vimos, ou seja, esse código não pertence nem a controles e nem a modelos. Por essa razão, o Rails disponibiliza um local para você colocar esse tipo de lógica: *view helpers*, ou conhecido apenas como *helpers*.

Cuidado com código complexo em helpers!

Lembre-se que código complexo pode ser um mal sinal, ainda pior quando eles estão em *helpers*. Se você encontrar *helpers* com muitas linhas de código, ou muito `if`, tome isso como um alerta -- você pode estar escrevendo regras de negócio na camada errada, ou ainda, tentando fazer muita coisa em um lugar só. As maneiras de resolver esse problema são mover regras de negócio para a sua camada, ou então criar classes que estruturam os dados de formas mais adequadas para os templates.

Criaremos então o primeiro *helper*. Os geradores do Rails criam um arquivo de `helper` por controle (por exemplo, `rooms_helper.rb` para o controle `RoomsController`) e é uma boa maneira de organizá-los. Porém, como vamos usar o *helper* em outros controles, vamos colocar no *helper* geral, o `ApplicationHelper`. Nesse *helper*, vamos colocar o resultado da extração do código do template como um método. Portanto, abra o arquivo `app/helpers/application_helper.rb`:

```
module ApplicationHelper
  def error_tag(model, attribute)
    if model.errors.has_key? attribute
      content_tag(
        :div,
        model.errors[attribute].first,
        class: 'error_message'
      )
    end
  end
end
```

Os métodos dos *helpers* são inclusos no mesmo contexto dos templates, portanto temos acesso aos modelos via variáveis de instância, da mesma forma que fazemos no template. Porém, usá-las é desaconselhado, já que geramos uma dependência do código que não fica clara. Dessa forma, passaremos o modelo como parâmetro, seguido pelo atributo que queremos verificar a existência de erros. O restante é o mesmo que fizemos quando direto no template, com a diferença de que, ao invés de gerar o HTML manualmente, usamos um *helper* do Rails para criar a tag.

Há uma outra maneira de resolver o problema: retornar o HTML como texto e imprimir o resultado no template, mas não é recomendado. A razão é que o Rails, por padrão, faz o *escaping* de todo o conteúdo que não é marcado como seguro, evitando injeção de código malicioso. Portanto, é mais seguro e mais simples usar os métodos do próprio Rails. O resultado visual é o mesmo, mas o template fica muito mais limpo:

```
<p>
  <%= f.label :full_name %>
  <%= f.text_field :full_name %>
  <%= error_tag @user, :full_name %>
</p>
```

Repetindo essa alteração no restante dos campos, o template `app/views/users/_form.html.erb` fica assim:

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    Há erros no formulário, por favor verifique.
  </div>
<% end %>

<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %>
    <%= f.text_field :full_name %>
    <%= error_tag @user, :full_name %>
  </p>
  <p>
    <%= f.label :location %>
    <%= f.text_field :location %>
    <%= error_tag @user, :location %>
  </p>
  <p>
    <%= f.label :email %>
    <%= f.text_field :email %>
    <%= error_tag @user, :email %>
  </p>
  <p>
    <%= f.label :password %>
    <%= f.password_field :password %>
```

```
    <%= error_tag @user, :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %>
    <%= f.password_field :password_confirmation %>
    <%= error_tag @user, :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %>
    <%= f.text_area :bio %>
    <%= error_tag @user, :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Salve o arquivo e envie o formulário, com erros, e veja o resultado do trabalho. Bem melhor, não é mesmo? Como exercício, adicione algumas validações no modelo `Room`, faça as alterações no formulário para usar o *helper* e veja o resultado.

Capítulo 7

Faça sua aplicação falar várias línguas

Lembrar-se que morreremos um dia é a melhor forma que conheço para evitar a armadilha de pensar que temos algo a perder.
– Steve Jobs

7.1 O processo de internacionalização (I18n)

Até então, todos os formulários estão em inglês, inclusive as mensagens de erro. Fizemos dessa forma de propósito, pois vamos usar o sistema de internacionalização (apesar de, em português, a palavra "internacionalização" possuir 19 letras, em inglês ela possui 20, *internationalization*, e portanto é comumente abreviada por I + 18 letras + n, ou i18n) do próprio Rails.

O sistema de I18n do Rails funciona com uma árvore de pares chave valor, sendo que a chave é um identificador de uma mensagem e o valor é o seu texto. Todo o

conjunto de mensagens possui uma chave raiz que é o uma sigla representando o idioma. Essa árvore é armazenada em um arquivo YAML. Vamos a um exemplo hipotético:

```
pt-BR:
  users:
    sign_up:
      success: Cadastro realizado com sucesso.
      failed: Não foi possível realizar o cadastro.
```

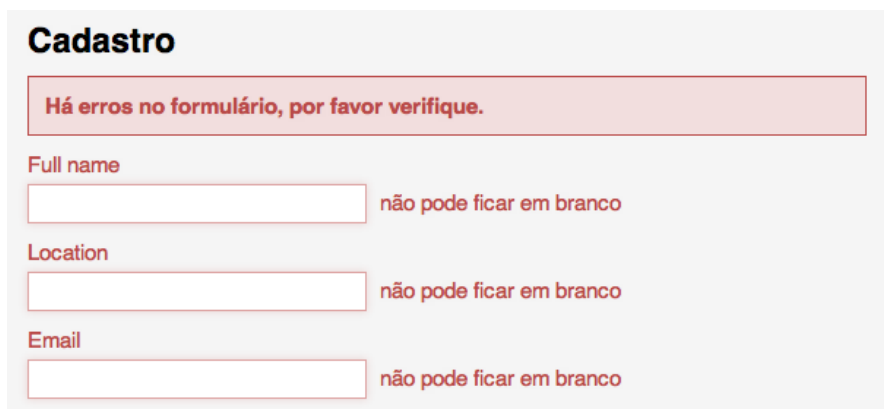
Essa árvore representa duas mensagens no idioma português, a mensagem de sucesso e a de erro. Para para acessar uma mensagem, basta montar uma string com a sequência dos passos da árvore. No caso da mensagem de sucesso, precisamos montar a string `"users.sign_up.success"`. Note que não mencionamos o idioma, pois isso é configurado de maneira global -- uma vez que configuramos o sistema de I18n com um idioma, todas as chaves a serem traduzidas usarão esse mesmo idioma.

Meu site só vai ser em português, preciso usar o I18n?

É considerada uma boa prática deixar seu sistema sempre traduzido com o I18n. A vantagem é que os templates ficarão muito mais limpos e mais fáceis de serem revistos por pessoas não técnicas.

Na pasta `config/locales` ficam os arquivos contendo as traduções. Se você abrir o arquivo `config/locales/en.yml`, que já vem com o Rails, você observará a seguinte estrutura:

```
en:
  hello: "Hello world"
```

Isso significa que, quando o sistema estiver usando o *locale* "en" (inglês), quando traduzirmos a chave `hello`, o resultado será "Hello world".

Os contribuidores do Rails já fizeram a tradução dos erros de validação para o português e portanto vamos usá-la como ponto de partida. Baixe o arquivo de tradução (vá para http://colcho.net/rails-i18n e clique em "Raw") e salve como `config/locales/rails.pt-BR.yml`. O nome do arquivo não importa, mas vamos separar nossas mensagens das mensagens do Rails para organização.

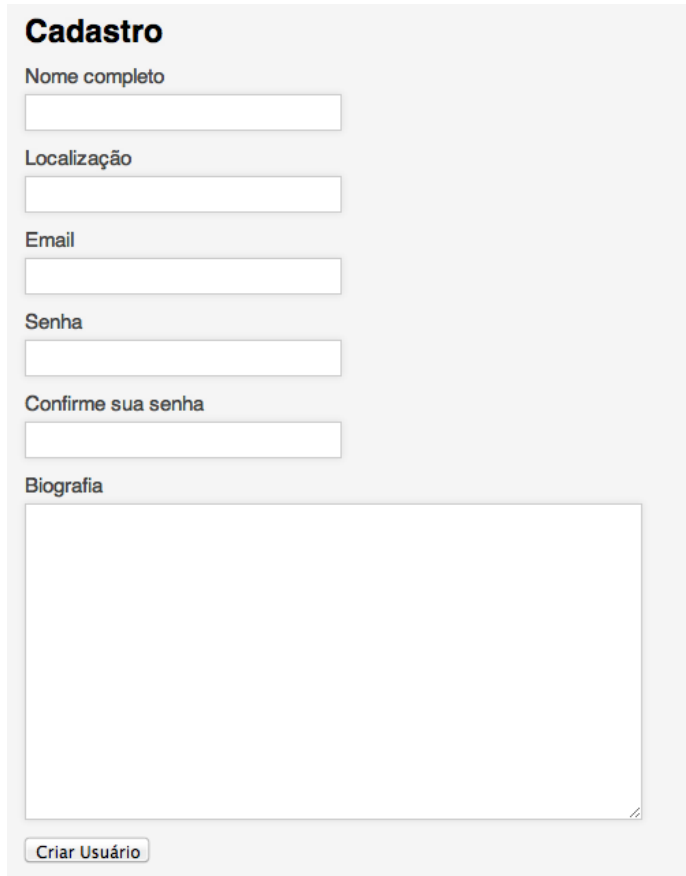
Após salvar o arquivo, vamos alterar o idioma padrão do Colcho.net para português brasileiro. Para isso, abra o arquivo config/application.rb e altere as linhas que estão comentadas por padrão para que fiquem da seguinte forma:

```
# Set Time.zone default to the specified zone
#    and make Active Record auto-convert to this zone.
# Run "rake -D time" for a list of
#   tasks for finding time zone names. Default is UTC.
config.time_zone = 'Brasilia'

# The default locale is :en and all
#   translations from config/locales/*.rb,yml are auto loaded.
# Rails.application.config.i18n.load_path
#   += Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
config.i18n.default_locale = :"pt-BR"
```

É necessário reiniciar o servidor para que o Rails possa carregar as novas configurações. Em seguida, vá a um dos formulários e tente enviar os dados em branco. *Et voilà*! Erros em português:

Figura 7.1: Erros traduzidos com o sistema de i18n

Os nomes dos atributos ainda continuam em inglês. Para traduzi-los, temos que seguir as convenções do Rails: activerecord.models.<modelo>, substituindo <modelo> pelo nome do modelo, de modo a traduzir o traduzir o botão "Create User", por exemplo. Para atributos, o caminho é um pouco mais longo: activerecord.attributes.<modelo>.<atributo>.

Vamos aplicar essas traduções ao modelo `User`, portanto crie o arquivo `config/locales/pt-BR.yml`:

```
pt-BR:
  activerecord:
    models:
      user: Usuário
    attributes:
      user:
        bio: Biografia
        email: Email
        full_name: Nome completo
        location: Localização
        password: Senha
        password_confirmation: Confirme sua senha
```

Em seguida, reinicie o servidor do Rails e atualize a página para ver os resultados. Vamos colocar os atributos também para quartos:

```
pt-BR:
  activerecord:
    models:
      room: Quarto
      user: Usuário
    attributes:
      user:
        bio: Biografia
        email: Email
        full_name: Nome completo
        location: Localização
        password: Senha
        password_confirmation: Confirme sua senha
      room:
        description: Descrição
        location: Localização
        title: Título
```

Quando o arquivo YAML já existe e apenas foi modificado, o Rails levará em conta as alterações, portanto não é necessário reiniciar o servidor. Assim, vá ao formulário de novos quartos e verifique se os campos foram traduzidos corretamente.

Figura 7.2: Formulário com atributos traduzidos

7.2 Traduza os templates

Agora que temos a integração de I18n com o `ActiveRecord`, vamos traduzir os templates. Para isso, vamos ter que chamar o mecanismo de I18n manualmente. Isso é feito com *helpers* `t` (atalho para `translate`), para tradução das chaves que já vimos, e `l` (atalho para `localize`) para a localização de datas, ou seja, mostrar uma data no formato do idioma em uso.

Vamos traduzir a página de cadastro, que possui apenas um título (`app/views/users/new.html.erb`). Trocamos o texto "Cadastro" pela tradução da chave `users.new.title`:

```
<h1><%= t 'users.new.title' %></h1>
<%= render 'form' %>
<%= link_to t('links.back'), root_path %>
```

Veja o resultado, mesmo sem criar a tradução no YAML. O Rails imprime a última chave numa tentativa de não deixar a página em branco. Porém, o Rails gera um HTML específico para chaves não traduzidas:

```
<h1>
  <span class="translation_missing"
        title="translation missing: pt-BR.users.new.title">
          Title
  </span>
</h1>
...
```

Para corrigir isso, vamos ao YAML de tradução (`config/locale/pt-BR.yml`) e adicionamos as chaves que acabamos de criar.

```
pt-BR:
  users:
    new:
      title: Cadastro

  links:
    back: Voltar

  activerecord:
  # continua ...
```

Ao recarregar a página, veremos o título e o link traduzidos corretamente.

Dica: estilo para a classe CSS “translation_missing”

No ambiente de desenvolvimento, podemos colocar um CSS específico para ajudar a identificar onde faltam elementos para serem traduzidos. Uma maneira de se fazer isso é criar um novo CSS, `app/assets/stylesheets/translation_missing.css.erb`, com o seguinte conteúdo:

```
<% if Rails.env.development? %>
  .translation_missing {
    border: 3px dashed red;
  }
<% end %>
```

Assim, textos a serem traduzidos possuirão uma marcação visual, sem impactar ambientes de produção ou testes.

Há uma melhoria que podemos fazer. O Rails automaticamente coloca o nome do controle e da ação no “escopo” das chaves de tradução caso você queira, bastando iniciar a chave com “.”. Por exemplo, se mudarmos o título para `.title`, a chave utilizada será `users.new.title`. Em um outro exemplo, no controle de quartos, ação `index`, a chave seria `rooms.index.title`. Vamos aplicar esse atalho:

```
<h1><%= t '.title' %></h1>
<%= render 'form' %>
<%= link_to t('links.back'), root_path %>
```

Fazendo o mesmo na ação *edit* (`app/views/users/edit.html.erb`):

```
<h1><%= t '.title' %></h1>
<%= render 'form' %>
<%= link_to t('links.back'), @user %>
```

Traduções para quartos

Os templates de quarto podem ser traduzidos da mesma maneira. Aproveite para fazer o mesmo nos outros templates de quarto: `app/views/rooms/new.html.erb` e `app/views/rooms/edit.html.erb`.

A ação *show* será um pouco diferente. Como no título temos o nome do usuário, precisaremos descobrir como passar parâmetros para o sistema de I18n. Isso é feito usando a notação `%{}`. Veja como fica o título dessa página:

```
pt-BR:
  users:
    new:
      title: Cadastro
    edit:
      title: Editar perfil
    show:
      title: "Perfil: %{user_name}"
      edit: 'Editar perfil'
      location: "Localização: %{location}"
      bio: "Bio: %{bio}"

  # ...
```

Veja que o uso de aspas é obrigatório para esse caso. Para passar os parâmetros, basta passar um hash mapeando os atributos a serem interpolados:

```
<h1><%= t '.title', user_name: @user.full_name %></h1>

<ul>
  <li><%= t '.location', location: @user.location %></li>
  <li><%= t '.bio', bio: @user.bio %></li>
</ul>

<%= link_to t('.edit'), edit_user_path(@user) %>
```

Por fim, vamos à *partial* de formulário (`app/views/users/_form.html.erb`):

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    <%= t 'general.form_error' %>
  </div>
<% end %>
...
```

O layout `(app/layouts/application.html.erb`):

```
...
      <nav>
        <ul>
          <li><%= link_to t('layout.rooms'), rooms_path %></li>
          <li><%= link_to t('layout.signup'), new_user_path %></li>
        </ul>
      </nav>
...
```

E no YAML (`config/locales/pt-BR.yml`):

```
# ...
links:
  back: Voltar

layout:
  rooms: Quartos
  signup: Cadastro

general:
  form_error: Há erros no formulário, por favor verifique.

activerecord:
# ...
```

Pronto, todos os templates estão prontos para receberem outros idiomas. No próximo capítulo, iremos trabalhar na última parte da funcionalidade de cadastro: o envio de e-mail de boas vindas e confirmação de conta. Porém, antes de chegarmos lá, vamos ver como trocamos o idioma de acordo com o usuário.

7.3 Extra: alterar o idioma do site

Vimos durante este capítulo como usar o mecanismo de internacionalização do Rails e configuramos o idioma padrão como português brasileiro, mas não vimos nenhuma maneira de usar outro idioma. Vamos deixar o Colcho.net apenas em português, mas é importante saber qual é a melhor forma de fazer seu site suportar mais de um idioma simultaneamente.

A maneira mais simples de suportar idiomas diferentes no seu site é especificando o idioma via URL. Por exemplo, para o idioma inglês, poderíamos criar a URL `www.colcho.net/en/users/new`.

A principal razão é o *caching*: Em geral, o *caching* é feito por URL e ter páginas diferentes (uma para cada idioma) em uma mesma URL pode tornar o *cache* mais difícil de ser construído. Quando o idioma faz parte da URL, não há necessidade de alterações no mecanismo de *caching*.

Para que as rotas com idiomas funcionem, usamos uma funcionalidade chamada `scope`. Os `scopes` em rotas servem para agrupar um conjunto de controles em um segmento de rota, como por exemplo, se criamos o recurso `rooms` no `scope` `admin`, a ação `new` mapeará para a URL `/admin/rooms/new`. Porém, esse recurso, usado essa maneira, não resolve nosso problema de uma maneira flexível: teríamos que criar uma entrada `scope` para cada idioma.

O que é um segmento de caminho?

Segmento de caminho (*path segments*) é uma parte da URL que é separada por "/" (como se fossem "pastas"). Por exemplo, na imagem a seguir, o caminho (*path*) é `/users/new`, sendo `users` e `new` segmentos distintos:

Figura 7.3: Algumas partes de uma URL

É possível fazer com que o escopo torne-se um parâmetro, e é exatamente isso que precisamos fazer para que possamos detectar, via URL, o idioma que o usuário está requisitando. Para isso, usamos a sintaxe de símbolos para a rota. Criamos o `scope` com o nome de `:locale` e, enfim, podemos acessar o conteúdo deste trecho da URL através da hash `params`:

```
Colchonet::Application.routes.draw do
  scope ":locale" do
    resources :rooms
```

```
    resources :users
  end

  root 'home#index'
end
```

Note que ao usar os `helpers` de rota do Rails, esse parâmetro será automaticamente incluído, portanto não é necessário alterar nenhum outro código: o idioma inicialmente selecionado pela URL.

A partir de agora, em todas as requisições, o que estiver na URL no segmento `:locale` será passado como parâmetro. Vamos trabalhar de forma a tornar essa variável disponível a todos os controles.

Tal como o `ActiveRecord`, os controles também possuem *callbacks*. Nesse caso, criaremos filtros, métodos que são executados antes de ações. Esses filtros, em geral, possuem duas naturezas: preparar dados para serem acessados nos controles ou afetar a navegação, por exemplo, bloquear acessos de requisições não autenticadas.

Filtros são criados através das *class macros*: `before_action`, para métodos executados antes da ação, `after_action` para depois, ou em volta, via `around_action`. O filtro mais usado é o `before_action`, e é ele que vamos usar. Concluindo, vamos criar um filtro no `ApplicationController` (`app/controllers/application_controller.rb`), para configurar o `locale` do sistema de internacionalização:

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception

  before_action do
    I18n.locale = params[:locale]
  end
end
```

> **MIGRANDO PARA RAILS 4: FILTROS**
>
> O `before_action` é um novo novo para os filtros de controle. Nas versões anteriores, os filtros eram conhecidos como `before_filter`, `after_filter` e `around_filter`. Eles ainda continuam funcionando, mas vamos usar a sintaxe mais moderna para o Rails 4.

Se acessarmos a URL `/pt-BR/users/new`, será possível ver o cadastro traduzido para o português. Substituindo `pt-BR` por `en`, veremos as páginas com marcações, pois não criamos todas as entradas para o inglês. Porém, após alterar o arquivo de rotas e o controle, não será mais possível acessar alguma das páginas, vamos encontrar erros. Isso porquê os *helpers* de URL do Rails agora precisam do *locale* para gerar corretamente.

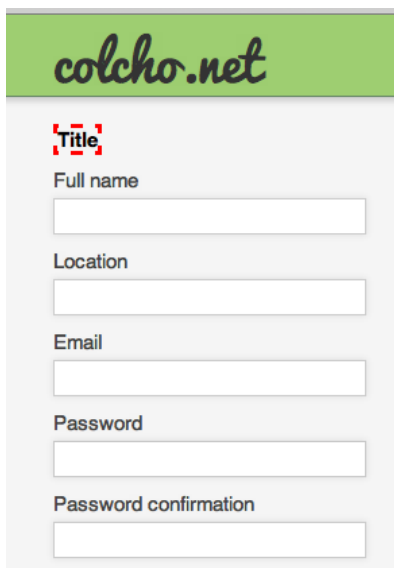



Figura 7.4: Site em inglês: marcação vermelha onde não temos tradução

Vamos continuar o desenvolvimento do site apenas no português.

Constraints em rotas

Um problema das rotas até este momento é que, se um usuário mudar o segmento

de idioma para algo que não temos tradução, como `jp`, verá um site incompleto ou não funcional. Para evitar esse problema, criaremos uma *constraint*, ou restrição, rota, de forma que a rota não seja válida no caso do idioma não ser suportado, exibindo ao usuário a mensagem de erro 404.

Podemos usar expressões regulares para criar restrições. Usaremos uma expressão regular que valida se o `:locale` é `en` ou `pt-BR`:

```
Colchonet::Application.routes.draw do
  scope ":locale", locale: /en|pt\-BR/ do
    resources :rooms
    resources :users
  end

  root "home#index"
end
```

Por fim, ainda existe um problema: se o usuário tentar acessar a *home*, ou rota raiz do site? O Rails não conseguirá criar as rotas para os quartos na listagem, e o motivo é simples: as rotas para os controles `RoomsController` e `UsersController` só existem se o parâmetro `:locale` existir e, no caso da *home*, ela não existe.

Segmentos opcionais nas rotas

Para solucionar o problema, vamos tornar a presença do `:locale` opcional, tornando o idioma padrão do site como português. Tomando como exemplo a rota de novo cadastro, teremos as seguintes possibilidades:

- `/users/new` - cadastro em português;
- `/pt-BR/users/new` - cadastro em português;
- `/en/users/new` - cadastro em inglês;
- `/jp/users/new` - erro "página não encontrada" (`jp`, `es`, ou qualquer outro idioma que não seja `pt-BR` ou `en`)

Para segmentos opcionais, o Rails usa a mesma notação de campos opcionais em expressões regulares:

```
Colchonet::Application.routes.draw do
  scope "(:locale)", locale: /en|pt\-BR/ do
```

```
    resources :rooms
    resources :users
  end

  root "home#index"
end
```

Ainda falta a rota na *home* que responde a diferentes idiomas. Ao acessar a *home*, ela será tratada como se não houvesse nenhum idioma, indo para o padrão português. Para que a raiz também responda a outros idiomas, temos que adicionar uma nova rota usando o `get`.

A rota do tipo `get` é o tipo de rota mais simples. Você pode desenhar qualquer rota e ainda usar a notação `":segmento"` para usar segmentos dentro do `params`. Adicionamos também a restrição de idiomas, extraindo a expressão regular para uma constante:

```
Colchonet::Application.routes.draw do
  LOCALES = /en|pt\-BR/

  scope "(:locale)", locale: LOCALES do
    resources :rooms
    resources :users
  end

  get '/:locale' => 'home#index', locale: LOCALES
  root "home#index"
end
```

> **Cuidado com rotas `get`**
>
> Tome cuidado com rotas `get`, pois elas deixam suas rotas mais complexas. Um tipo de bug comum ocasionado por esse tipo de rotas é quando elas entram em conflito, ou seja, mais de uma regra satisfaz uma URL. Nesse caso, o Rails sempre escolherá a primeira rota encontrada, de cima para baixo, no arquivo de rotas.

Um problema que essa rota opcional gera é que todos os links da sua aplicação teriam que passar a opção `locale: I18n.locale` para **todos** os *helpers* de rota, já

que ela pode não existir. O Rails consegue centralizar essa opção se implementarmos um método chamado `default_url_options`.

Precisamos também atualizar o filtro para que, caso o `:locale` não esteja definido, vamos usar o idioma configurado como padrão no arquivo `config/application.rb`.

Por fim, o `ApplicationController` (`app/controllers/application_controller.rb`) deverá ficar da seguinte forma:

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception

  before_action do
    I18n.locale = params[:locale] || I18n.default_locale
  end

  def default_url_options
    { locale: I18n.locale }
  end
end
```

Essas funcionalidades usam alguns recursos que requerem bastante conhecimento do framework, portanto não se assuste se você achou complicado. Rotas e links podem se tornar complexas em aplicações de grande porte, portanto existem diversas ferramentas para dar suporte a este tipo de requisito.

Note também que existem outras maneiras de detectar o idioma, como por exemplo via diferentes domínios, via *headers* HTTP ou até GeoIP, um banco de dados que, através do IP do usuário, determina-se o país e por consequência, seu idioma. Essas opções não serão abordadas dada a sua complexidade. Se você quiser mais informações, verifique o guia oficial do Rails no assunto: http://guides.rubyonrails.org/i18n.html.

Capítulo 8

O cadastro do usuário e a confirmação da identidade

"Suor mais sacrifício é igual a sucesso"
– Charles Finley

8.1 Entenda o ActionMailer e use o MailCatcher

O Rails possui uma ferramenta de envio de emails chamada `ActionMailer`. Ela funciona como se fosse um controle, ou seja, possui diversos pontos de entrada (em análogo com as `actions`) e resulta em um template que será enviado por email para o usuário.

Para facilitar o desenvolvimento da entrega de emails, usaremos uma ferramenta chamada `mailcatcher`. O `mailcatcher` é uma gem Ruby que disponibiliza dois serviços: um servidor de envio de emails SMTP, que será usado pelo Rails, e um servidor Web, no qual observamos os emails que foram enviados, como uma caixa

de entrada. Nela poderemos ver as versões HTML, texto e os dados brutos que foram enviados pelo Rails.

Para instalá-la, basta executar:

```
$ gem install mailcatcher
 ...
Successfully installed mailcatcher-0.5.12
8 gems installed
```

Ao executar o comando `mailcatcher`, temos:

```
$ mailcatcher
Starting MailCatcher
==> smtp://127.0.0.1:1025
==> http://127.0.0.1:1080
*** MailCatcher now runs as a daemon by default. Go to the web interface
to quit.
```

Com o `mailcatcher` em execução, vamos voltar ao Rails. Criaremos um *mailer* chamado `Signup`. Essa classe irá agrupar todos os emails relacionados com o cadastro (*sign up*) e é essa a natureza dos *mailers*: cada classe é um contexto comum a uma série de emails. Um outro exemplo de *mailer* poderia ser `Notifications`, no qual todos os emails de notificação, como um um novo anúncio de quarto, um novo comentário ou uma nova avaliação.

Generator para mailers

O Rails possui geradores para Mailers, mas eles são bem simples. Por esse motivo, vamos criar manualmente.

O `Signup` irá herdar de `ActionMailer::Base`, que possui um comportamento muito parecido com o `ActionController::Base`. Herdar dessa classe irá adicionar diversos métodos facilitadores para criar emails. Por exemplo, poderemos colocar valores padrão, como o campo “De:”, ou “From:”.

Cada método criado no `Signup` funciona equivalentemente com as *actions* de um controle, ou seja, no caso dos emails de confirmação de conta, criaremos um método chamado `confirm_email` que será responsável pela montagem do email. Outro paralelo com os controles é a forma de como comunicamos variáveis com o

template: através de variáveis de instância. Por fim, os métodos deverão retornar um objeto `Mail`. Para isso, chamamos o método `mail`, passando a ele os campos de email, como assunto (*subject*), destinatário (*to*), entre outros.

Vamos então criar o email de confirmação. Nele, teremos o *link* de confirmação, que vamos criar depois, o destinatário (*to*) e o título (*subject*) do email. Vamos também usar um truque bem útil: colocar um email de nosso controle como *bcc*, ou seja, cópia oculta.

O resultado do `Signup` (`app/mailers/signup.rb`) é o seguinte:

```
class SignupMailer < ActionMailer::Base
  default from: 'no-reply@colcho.net'

  def confirm_email(user)
    @user = user
    # Link temporário pois a funcionalidade ainda
    # não existe, vamos criar ainda neste capítulo
    @confirmation_link = root_url

    mail({
      to: user.email,
      bcc: ['sign ups <signups@colcho.net>'],
      subject: I18n.t('signup.confirm_email.subject')
    })
  end
end
```

Emails via BCC

É uma boa ideia enviar emails para você mesmo usando o BCC. Recebendo esses emails, você pode ver exatamente o que o usuário vê, para tentar ajudá-lo caso haja algum problema técnico ou na experiência de usuário.

Outro uso do BCC é para envio de emails em massa, para evitar que os emails de todos os usuários fiquem à mostra.

8.2 Templates de email, eu preciso deles?

É boa prática enviar conteúdo HTML e texto puro em um mesmo email. Assim, fica a critério do seu usuário a maneira que ele prefere ler emails. Portanto é necessário

criar dois templates para cada "ação". Ao seguir a convenção do Rails, ele já faz o trabalho de criar um email que contém as duas representações (chamado de *multipart*) e enviar para o destinatário.

Crie a pasta `signup` na pasta `app/views` e lá vamos criar o template `confirm_email.html.erb`, a versão HTML do email:

```
<h1><%= t '.title' %></h1>
<p>
  <%= t '.body', full_name: @user.full_name %>
</p>
<p>
  <%= t '.confirm_link_html',
      link: link_to(@confirmation_link, @confirmation_link) %>
</p>
<p>
  <%= t '.thanks_html', link: link_to('Colcho.net', root_url) %>
</p>
```

Agora vamos fazer o template para o email em texto puro. Crie o arquivo `app/views/signup/confirm_email.text.erb` com o seguinte conteúdo:

```
<%= t '.title' %>

<%= t '.body', full_name: @user.full_name %>
<%= t '.confirm_link_html', link: @confirmation_link %>

<%= t '.thanks_html', link: root_url %>
```

Atenção aos links nos mailers

Preste atenção nos links que estamos criando nos templates de *mailer*. Estamos usando o sufixo **_url** e não o **_path**. Quando estamos na Web, o browser infere o servidor e por isso só o `path` é suficiente, mas não no caso de emails. Essa confusão porque sempre usamos **_path** nos templates, portanto é comum confundirmos.

Para os textos traduzidos, adicione as seguintes linhas no YAML do idioma português (`config/locales/pt-BR.yml`), tomando o cuidado para colocar no nível correto, já que a identação é importante:

```
pt-BR:
  # ...

  signup:
    confirm_email:
      subject: 'Colcho.net - Confirme seu email'
      title: 'Seja bem vindo ao Colcho.net!'
      body: |
          Seja bem vindo ao Colcho.net, %{full_name}.
          O Colcho.net é o lugar ideal para você alugar aquele quarto
          sobrando na sua casa e ainda conhecer gente do mundo inteiro.

      confirm_link_html: 'Para você começar a usar o site, acesse o
              link: %{link}'
      thanks_html: 'Obrigado por se cadastrar no %{link}.'

  links:
  # ...
```

Chaves com sufixo _html

Na seção 6.10 falamos de *escaping* de tags HTML e injeção de código malicioso. O sistema de I18n também se preocupa com isso. Então, para evitar que todos os seus links virem texto, é necessário colocar o sufixo `_html` nas chaves de tradução, para que o texto final seja um HTML válido.

Assumindo que temos um usuário cadastrado no banco de dados, vamos tentar enviar um email para ele:

```
Signup.confirm_email(User.first)
# ArgumentError: Missing host to link to!
#   Please provide the :host parameter,
#   set default_url_options[:host], or set :only_path to true
```

Atenção: chamadas de método na classe

Em *mailers*, mesmo definindo métodos de instância, devemos chamá-los diretamente na classe, já que o *mailer* é construído de acordo com o método a ser chamado.

O problema está na hora de gerar links. Pode parecer simples, mas para o Rails saber em que endereço ele está não é uma tarefa trivial. Na verdade, ele não tem como saber, dadas as maneiras que um servidor pode ser configurado. Por isso, o que o Rails faz é confiar nos cabeçalhos HTTP (vários deles, dependendo de onde veio a requisição) para montar a URL completa.

O problema é que, frequentemente, os mailers são executados fora do contexto de requisição web, ou seja, em uma tarefa Rake, ou em *jobs* em segundo plano, e, portanto, impossível saber o host do servidor. Dessa forma, o Rails requer que você, manualmente, defina o host para essas situações. Essa configuração pode ser definida por cada ambiente (desenvolvimento, teste e produção). Vamos corrigir para o nosso ambiente de desenvolvimento.

Para isso, abra o arquivo `config/environments/development.rb` e adicione a linha relacionada a `default_url_options`, antes do `end` no final:

```
config.assets.debug = true

  # Aponta o host para o ambiente de desenvolvimento
  config.action_mailer.default_url_options = {
    host: "localhost:3000"
  }
end
```

Após reiniciar o console, conseguimos enviar o email:

```
mail = Signup.confirm_email(User.first)
# Rendered signup_mailer/confirm_email.html.erb (1.5ms)
# Rendered signup_mailer/confirm_email.text.erb (1.7ms)
# => #<Mail::Message:70285182528740, ... >

irb(main):003:0> mail.deliver
# Conteúdo do email...
```

Objetos criados e email enviado para o limbo, mas isso não significa muita coisa. Vamos configurar o Rails para que ele possa enviar emails de verdade, usando o serviço de SMTP do `mailcatcher`.

Para isso, abra novamente o `config/environments/development.rb` e coloque as seguintes linhas abaixo da linha que adicionamos anteriormente:

```
# Aponta o host para o ambiente de desenvolvimento
  config.action_mailer.default_url_options = {
    host: "localhost:3000"
  }

  config.action_mailer.delivery_method = :smtp
  config.action_mailer.smtp_settings = {
    address: "localhost",
    port: 1025
  }
end
```

Reinicie o console do Rails e tente novamente:

```
Signup.confirm_email(User.first).deliver
# => #<Mail::Message:70316580535500, Multipart: true, ...>
```

Aponte o seu browser para o endereço web do `mailcatcher` (http://localhost:1080) e veja seus emails. Note que você pode ver também a versão puro texto. Agora que conseguimos enviar emails, vamos plugar essa classe no momento do cadastro do usuário.

Precisamos alterar o controle `UsersController` e, no momento que uma conta é criada com sucesso, disparamos o email. Vá ao controle e adicione a linha logo após o `if`:

```
class UsersController < ApplicationController
  # ...

  def create
    @user = User.new(params[:user])
    if @user.save
      Signup.confirm_email(@user).deliver

      redirect_to @user, notice: 'Cadastro criado com sucesso!'
    else
```

```
      render :new
    end
  end

  # ...
end
```

Lembre-se de reiniciar o servidor do Rails, caso não o tenha feito após as alterações no arquivo `config/environments/development.rb`. Ao fazer um cadastro, o email será disparado e capturado pelo `mailcatcher`.

Terminamos o envio de email de confirmação. Lembre-se que colocamos um link temporário no email e portanto, vamos corrigi-lo, colocando a implementação correta.

8.3 Mais emails e a confirmação da conta de usuário

Mesmo ainda não tendo login verificado por email e senha, vamos limitar o acesso a contas não confirmadas. Vamos deixar tudo pronto para implementarmos o login completo no próximo capítulo.

Criaremos agora a última parte que resta para terminar a funcionalidade de cadastro, que é a confirmação de email. O que devemos fazer é: enviar o email ao usuário e, quando ele clicar em um link único para a conta dele, a conta estará confirmada. Faremos isso gerando um token único para cada usuário. Quando o usuário for à pagina de confirmação com o token que foi incluído no email, podemos assumir que o usuário recebeu o email e está clicando no link.

Para implementar essa ideia, vamos criar dois novos campos no modelo Usuário: **confirmation_token** e **confirmed_at**. O primeiro campo, `confirmation_token`, será gerado automaticamente no momento do cadastro do usuário. Em seguida, enviamos o email (que acabamos de fazer) com um link, contendo o token gerado. O *token* tem que ser grande e aleatório, de maneira que seja praticamente impossível ser adivinhado e com grande probabilidade de ser único por usuário. O segundo é o `confirmed_at`, o campo que vamos marcar quando o usuário seguir o link recebido no email, tornando o cadastro do usuário totalmente válido.

Para isso, vamos gerar uma nova migração:

```
$ rails generate migration add_confirmation_fields_to_users \
      confirmed_at:datetime confirmation_token
```

```
invoke  active_record
create  db/migrate/
           20130719061229_add_confirmation_fields_to_users.rb
```

A migração gerada será:

```
class AddConfirmationFieldsToUsers < ActiveRecord::Migration
  def change
    add_column :users, :confirmed_at, :datetime
    add_column :users, :confirmation_token, :string
  end
end
```

Muitas adições em uma migração

Se você quiser adicionar muitas colunas a uma tabela, pode usar o `change_table`. Com o `change_table`, você informará o nome da tabela apenas uma vez e usará a mesma sintaxe do `create_table`. A migração anterior ficaria da seguinte forma:

```
change_table :users do |t|
  t.datetime :confirmed_at
  t.string :confirmation_token
end
```

Execute as migrações:

```
$ rake db:migrate
==  AddConfirmationFieldsToUsers: migrating ============
-- add_column(:users, :confirmed_at, :datetime)
   -> 0.0014s
-- add_column(:users, :confirmation_token, :string)
   -> 0.0004s
==  AddConfirmationFieldsToUsers: migrated (0.0020s) ===
```

8.4 Um pouco de callbacks para realizar tarefas pontuais

Agora vamos gerar o *token*. Para isso, usaremos *callbacks* do `ActiveRecord`. O `ActiveRecord` possui uma série de eventos durante o processo de salvar um objeto. Por exemplo, ao **criar** um objeto, a ordem de chamada é a seguinte:

1) `before_validation` - Antes da validação

2) `validate` - Executa as validações no modelo

3) `after_validation` - Após todas as validações

4) `before_save` - Antes de salvar

5) `before_create` - Antes de criar

6) `create` - Executa a criação do modelo

7) `after_create` - Depois de criar

8) `after_save` - Depois de salvar

9) `after_commit` - Depois de finalizar a transação no banco de dados

O mesmo acontece para atualização de um modelo, a diferença é que, ao invés de usar a palavra-chave `create`, usamos `update`. Em cada um desses eventos é possível plugar código para que possamos executar alguma operação. Vamos usar o evento `before_create` para preencher o *token* automaticamente.

Cuidado com os callbacks

É muito conveniente usar *callbacks*, mas é **muito** importante ter cautela por inúmeras razões. A primeira é que, se você fizer muitas operações, salvar um modelo torna-se algo demorado. A segunda é que você pode acabar gerando *loops* infinitos e situações complicadas de depurar. Tome bastante cuidado e faça coisas muito simples nesses *callbacks*. Se precisar de lógicas complexas, crie métodos e chame-os quando adequado.

No modelo `User` (`app/models/user.rb`), vamos adicionar o código responsável para gerar o *token*:

```
class User < ActiveRecord::Base
  # Omitindo conteúdo anterior...

  has_secure_password

  before_create do |user|
    user.confirmation_token = SecureRandom.urlsafe_base64
  end
end
```

O `before_create` aceita símbolo com o nome do método a ser chamado ou um um bloco associado, como fizemos nesse caso. No bloco, usamos a biblioteca `SecureRandom`, que gera números aleatórios o suficiente para nosso uso. Vamos usar também uma forma que seja possível ser passado via URLs e o `SafeRandom` já possui um método para isso. Lembre-se de não chamar o método `#save`, pois a gravação do objeto será realizada em um outro momento, assim que todos os *callbacks* estiverem concluídos. Ao chamar o método `#save` em *callbacks*, você gerará um loop infinito.

Para testar, crie um novo usuário via o browser e depois vá ao console do Rails:

```
User.last.confirmation_token
# => "o8iIOCdUzIXjivrLsfUA8g"
```

> **Não adicione o confirmation_token ao permit**
>
> Este é um campo que, se permitirmos ser atualizado via formulário, mesmo que de forma indireta, podemos prejudicar a segurança da aplicação, portanto não permita que o campo `confirmation_token` seja passível de *mass assignment*.

Criaremos o método `#confirm!`, que marca a data e a hora da confirmação e limpa o *token* do usuário, de forma que o link de confirmação só funcione uma única vez. Vamos também criar o `#confirmed?`, que verificará se a conta do usuário foi confirmada:

```
class User < ActiveRecord::Base
  # ...

  before_create do |user|
    user.confirmation_token = SecureRandom.urlsafe_base64
  end


  def confirm!
    return if confirmed?

    self.confirmed_at = Time.current
    self.confirmation_token = ''
    save!
  end

  def confirmed?
    confirmed_at.present?
  end
end
```

O método `confirm!` marca o campo `confirmed_at` com a hora corrente, limpa o *token* e salva o modelo. Experimente no console:

```
user = User.last
# => #<User id: 1, full_name: "Vinicius Baggio Fuentes", ...>

user.confirm!
# => true

user.confirmed?
# => true

user.confirmation_token
# => ""
```

Com este código, estamos quase prontos. Os últimos passos são: criar a rota, o controle e corrigir o link no *mailer*.

8.5 Roteamento com restrições

Vamos adicionar uma nova rota no arquivo `config/routes.rb`:

```
Colchonet::Application.routes.draw do
  resources :rooms
  resources :users

  resource :confirmation, only: [:show]

  root "home#index"
end
```

Note que há uma diferença nesse recurso: não usamos `resources`, mas sim `resource`, no singular. O motivo é que, para quem está navegando (ou o cliente de uma API), não existe mais de uma confirmação (ou seja, não faz sentido existir uma ação *index*) no sistema. Isso é chamado de *recurso singleton*. Em seguida, passamos uma opção, que é `:only => [:show]`. Isso significa que só queremos que a ação *show* seja criada, que é a ação que mais se aproxima do que queremos fazer.

Para entender o que essa rota gera, podemos executar o comando `rake routes` na raiz do projeto:

```
$ rake routes
...

confirmation GET  /confirmation(.:format) confirmations#show
```

Vamos criar o controle `ConfirmationsController`. Note que ainda temos que usar o plural no nome e o motivo é que você ainda pode usar o mesmo controle para rotas não-singleton. Nesse controle, vamos buscar o usuário a partir do token e, se ele de fato existir, confirmaremos a conta e então redirecionamos o usuário para a sua página do perfil. Então criemos o arquivo `app/controllers/confirmations_controller.rb`, com o seguinte conteúdo:

```
class ConfirmationsController < ApplicationController
  def show
    user = User.find_by(confirmation_token: params[:token])

    if user.present?
      user.confirm!
      redirect_to user,
        notice: I18n.t('confirmations.success')
    else
```

```
      redirect_to root_path
    end
  end
end
```

O `find_by` aceita um hash com os parâmetros para fazer a busca, e retornará o primeiro resultado encontrado. Adicionaremos a chave `confirmations.success` no YAML de traduções (`config/locales/pt-BR.yml`):

```
pt-BR:
  confirmations:
    success: Email confirmado com sucesso, obrigado!

  # ...
```

> **E OS DYNAMIC FINDERS?**
>
> Se você leu a primeira edição do livro, ou já programa em Rails 3, notará que a mesma busca pode ser feita com o método `find_by_confirmation_token`. Essa sintaxe foi extraída do Rails e se tornará obsoleta na próxima versão, portanto use escopos ou o `find_by`.

E finalmente, vamos corrigir o link no *mailer* do cadastro, que tínhamos deixado como `root_path` (`app/mailers/signup.rb`):

```
class Signup < ActionMailer::Base
  default from: 'no-reply@colcho.net'

  def confirm_email(user)
    @user = user
    @confirmation_link = confirmation_url({
      token: @user.confirmation_token
    })

    mail({
      to: user.email,
      bcc: ['sign ups <signups@colcho.net>'],
```

```
      subject: I18n.t('signup.confirm_email.subject')
    })
  end
end
```

Pronto, terminamos! Ao fazer um cadastro, recebemos um e-mail no `mailcatcher`:

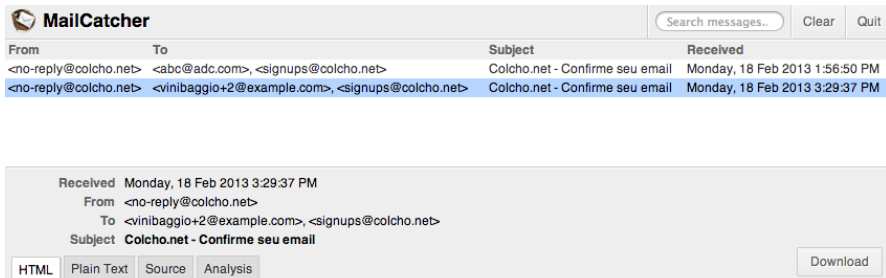



Figura 8.1: Email de confirmação no mailcatcher

Ao seguir o link no email, temos o seguinte resultado:

Figura 8.2: Email confirmado

Pronto, o cadastro está funcionando por completo!

8.6 Em resumo

Passamos os últimos capítulos exclusivamente desenvolvendo a funcionalidade de cadastro. Veja o que você aprendeu até agora:

- **Comandos** - Comandos básicos do Rails e do rake, como geradores, visualizar rotas, executar servidor e console;
- **Migrações** - Como gerar migrações que criam tabelas, adicionam índices e colunas;
- **Modelos** - Como criar modelos, incluindo validações e *callbacks*;
- **Segurança** - Como evitar injeção de código malicioso via *helpers*, como evitar ataques via *mass assignment*, como encriptar senha dos usuários e o que é *Cross Site Request Forgery*;
- **Emails** - Envio de emails usando *multipart*, enviando HTML e texto puro ao mesmo tempo;

- **Rotas** - Como construir rotas simples;
- **Asset Pipeline** - Aprendeu como um dos componentes mais complexos do Rails funciona e como trabalhar com ele;
- **Controle** - Controle de fluxo de navegação e mensagens *flash*;
- **I18n** - Internacionalização das mensagens da aplicação;
- **Templates e Helpers** - Criação de templates, layouts, *partials*, *view helpers* e deixar seus templates elegantes;

A lista é enorme, parabéns! Vamos agora usar um pouco de tudo para construir a funcionalidade de login.

Capítulo 9

Login do usuário

Foco é uma questão de decidir as coisas que você não irá fazer.
– John Carmack, cofundador da idSoftware

Um usuário agora pode se cadastrar, confirmando sua conta através do link enviado no email de boas vindas. Porém, ainda não temos nenhum controle de sessão, ou seja, qualquer visitante pode alterar qualquer informação no site.

Construiremos então o recurso "sessão", que será criado toda vez que um usuário realizar o login. A sessão do usuário durará todo o tempo de navegação, até que o usuário faça o "logout", ou seja, destruir o recurso "sessão". Criaremos os *templates* e o controle correspondente a esse fluxo de navegação e vamos usar a autenticação provido pelo `has_secure_password`.

Por fim, vamos modificar o cadastro de quartos para que quartos sejam associados ao criador e criar permissões, de forma que apenas o criador do quarto possa remover ou atualizar um quarto.

9.1 Trabalhe com a sessão

Para que o usuário só preencha o login apenas uma vez, precisamos guardar essa informação de alguma forma. A maneira que usamos para guardar estado entre requisições, ou seja, saber que alguma ação foi feita anteriormente, é usando *cookies*, dados que são "anexados" a todas as requisições. O Rails usa *cookies* para gravar a **sessão**.

Usaremos a sessão do Rails para gravar dados de modo a indicar que as requisições seguintes pertencem a um usuário específico. Usamos a sessão como um hash, ou seja, um conjunto chave-valor. Por exemplo, podemos indicar que o `user_id` daquela sessão é o número hipotético 42. Assim, antes de realizar qualquer processamento na aplicação, buscamos no banco de dados, o usuário com `id` 42.

Falando de recursos, a criação do recurso sessão será feita quando o usuário envia suas credenciais. Quando elas estiverem corretas, escreveremos o ID do usuário na sessão. Por sua vez, quando destruirmos a sessão, limparemos esse dado.

Como garantir a segurança das sessões?

É importante lembrar que os *cookies* são gravados no browser do usuário. Isso significa que é possível que um usuário veja todo o conteúdo dos *cookies*, ou seja, poderá ver seu `user_id`. O que impedirá, portanto, que algum usuário malicioso forje o *cookie* com o ID de outra pessoa?

O Rails assina digitalmente os *cookies* gravados no browser com uma chave que é secreta -- apenas o servidor sabe e nunca é compartilhado com os clientes (veja a chave no arquivo `config/initializers/secret_token.rb`). Quando um usuário malicioso altera ou cria um *cookie*, a sessão se tornará inválida, pois não será assinada com a chave, e o Rails irá rejeitar a sessão.

Note que, mesmo que seja difícil criar sessões forjadas, ainda é possível ler o conteúdo, portanto lembre-se de nunca gravar dados importantes na sessão, de preferência apenas IDs numéricos, por exemplo.

Começaremos criando a classe `UserSession`. Essa classe vai possuir as seguintes responsabilidades:

- Tradução de atributos (via I18n);

- Validações em formulário;
- Verificar as credenciais do usuário (email e senha);
- Gravar a sessão do usuário nos *cookies*.

Para tradução de atributos e validações de usuários, vamos usar um componente do Rails chamado `ActiveModel`. Ele contém a lógica de *callbacks*, validações e traduções do `ActiveRecord`, além de outras coisas. Além disso, o `ActiveModel` define a interface que modelos fazem com as outras camadas, como por exemplo, determinar rota e métodos auxiliares de construção de formulários.

Mesmo não gravando os dados do `UserSession` em banco de dados, podemos usar o `ActiveModel` para que essa classe interaja bem com as outras camadas do Rails e usufruir dos mecanismos de validação, bastante conveniente para a construção de formulários.

O `ActiveModel` possui uma lista grande de componentes úteis, alguns exemplos interessantes são:

- `ActiveModel::Callbacks` - *Callbacks* no ciclo de vida, na forma que vimos com o `ActiveRecord`;
- `ActiveModel::Conversion` - Usado para detectar templates e rotas;
- `ActiveModel::Dirty` - Suporte a atributos "sujos", ou seja, detectar quando um atributo foi alterado e guardar o valor antigo;
- `ActiveModel::Naming` - Usado para criar nomes legíveis para humanos, além de rotas;
- `ActiveModel::Translation` - Usado para traduzir atributos com o I18n;
- `ActiveModel::Validations` - Validação de atributos (unicidade, presença, etc.).

Alguns componentes são mais focados na integração com o `ActiveRecord` (como o `ActiveModel::AttributeMethods`), útil para desenvolvedores que estão fazendo bibliotecas e querem maior integração com o Rails (como por exemplo, suporte ao MongoDB (http://mongodb.org) , uma forma diferente de armazenar dados). Na documentação de cada componente existe como você deve usá-lo e os benefícios.

Novo no Rails 4, usaremos o `ActiveModel::Model`, que nos dá as seguintes funcionalidades:

- Validações;
- Integração com formulários e rotas;
- Traduções
- Inicialização de objetos com hashes;

Esse componente facilita muito a criação de modelos que interagem com o Rails mas que não mapeiam diretamente para modelos `ActiveRecord`.

Dica de uso do ActiveModel::Model

Algumas vezes é necessário criar um formulário complicado, envolvendo mais de um modelo, por exemplo. Requer bastante esforço realizar essa tarefa sem resultar em um código de qualidade questionável.

A dica é criar um modelo usando `ActiveModel::Model`. Com esse modelo, você pode criar atributos virtuais que mapeiam para cada campo do formulário, inclusive com validações. Uma vez o formulário construído, você pode criar código dentro da classe que irá resultar na gravação de todos os outros modelos envolvidos, deixando o código dos templates muito mais simples.

Vamos criar a classe `UserSession`. Nela, vamos usar validações, como se estivéssemos lidando com um modelo `ActiveRecord`. Declararemos os atributos `email` e `password`. Crie então o arquivo `app/models/user_session.rb`:

```
class UserSession
  include ActiveModel::Model

  attr_accessor :email, :password
  validates_presence_of :email, :password
end
```

Usando o console do Rails, vamos observar o comportamento da classe `UserSession`, com o que temos até agora:

```
session = UserSession.new
# => #<UserSession:0x007fee1d3b72a8>

session.valid?
# => false

session.errors.full_messages
# => ["Email não pode ficar em branco",
#     "Password não pode ficar em branco"]

session = UserSession.new(email: 'vinibaggio@example.com',
                          password: 'segredo')
# => #<UserSession:0x007ffba4d8c1e0
#        @email="vinibaggio@example.com",
#        @password="segredo">

session.valid?
# => true
```

Com poucas linhas de código, temos um comportamento bastante complexo, graças a modularidade do Rails 4. Ainda há muito o que se trabalhar nessa classe, porém primeiro vamos criar os templates e o controle, para que possamos testar a parte de autenticação a partir da tela.

9.2 Controles e rotas para o novo recurso

Vamos então criar as rotas do recurso, que haverão apenas as ações `create`, `new` e `destroy`. Para isso, criaremos mais uma entrada no arquivo `config/routes.rb`:

```
Colchonet::Application.routes.draw do
  resources :rooms
  resources :users

  resource :confirmation, only: [:show]
  resource :user_sessions, only: [:create, :new, :destroy]

  root "home#index"
end
```

Em seguida, criamos o controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`) e vamos mostrar

o formulário de login. Usamos a mesma técnica que usamos com modelos normais: criamos a variável no controle e a usamos no template:

```
class UserSessionsController < ApplicationController
  def new
    @user_session = UserSession.new
  end

  def create
    # Ainda não :-)
  end

  def destroy
    # Ainda não :-)
  end
end
```

Vamos criar o template para a ação `new`, bem similar aos formulários que fizemos até então. A única diferença é que vamos mostrar erros que não estão em um atributo em si, mas no objeto como um todo. Quando um usuário digita um email ou senha incorretamente, devemos mostrar que as credenciais estão inválidas. Fazemos isso por medida de segurança: não deixamos transparecer ao usuário se o email é válido ou não. Para isso, adicionamos o erro no "atributo" `:base`.

Precisamos criar a pasta `app/views/user_sessions` e o arquivo `%app/views/user_sessions/new.html.erb`:

```
<h1><%= t '.title' %></h1>

<% if @user_session.errors.has_key? :base %>
  <div id="error_explanation">
    <%= @user_session.errors[:base].join(', ') %>
  </div>
<% end %>

<%= form_for @user_session do |f| %>
  <p>
    <%= f.label :email %>
    <%= f.text_field :email %>
    <%= error_tag @user_session, :email %>
  </p>
  <p>
```

```
    <%= f.label :password %>
    <%= f.password_field :password %>
    <%= error_tag @user_session, :password %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<p><%= link_to t('.sign_up'), new_user_path %></p>
```

Ao direcionar o browser na página de nova sessão de usuário (http://localhost:3000/user_sessions/new), temos o seguinte:

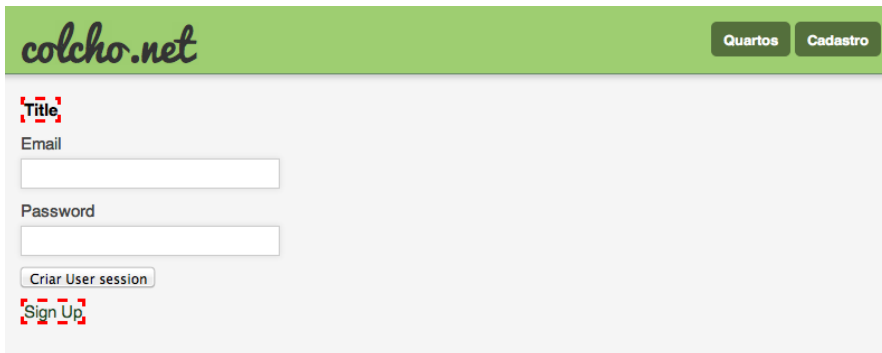



Figura 9.1: Página de login

As bordas vermelhas estão gritando, então vamos criar as traduções que estão faltando no arquivo I18n (`config/locales/pt-BR.yml`).

```
# ...
  show:
    edit: 'Editar perfil'
    title: "Perfil: %{user_name}"
    location: "Localização: %{location}"
    bio: "Bio: %{bio}"
# Nova parte:
user_sessions:
  new:
    title: 'Login'
```

```
    sign_up: 'Cadastre-se'

signup:
  confirm_email:
#...
```

No fim do arquivo, vamos criar mais duas "seções", contendo as traduções do `ActiveModel` e a tradução específica do botão de login:

```
room:
      description: Descrição
      location: Localização
      title: Título

# Adicione:
activemodel:
  attributes:
    user_session:
      email: Email
      password: Senha
  errors:
    messages:
      invalid_login: 'Usuário ou senha inválidos'

helpers:
  submit:
    user_session:
      create: 'Entrar'
```

Note a porção final do I18n, um trecho especial para os formulários. Você pode criar uma tradução específica para cada modelo como fizemos, usando as chaves adequadas. O padrão que abrange todos os modelos é o seguinte, extraído da tradução padrão do Rails que baixamos (`config/locales/rails.pt-BR.yml`):

```
# Não coloque no seu pt-BR.yml,
# você já tem isso no rails.pt-BR.yml
pt-BR:
  # ...
  helpers:
    submit:
      create: Criar %{model}
```

```
submit: Salvar %{model}
update: Atualizar %{model}
```

É interessante dar uma olhada no I18n padrão do Rails caso você tenha interesse em customizar uma mensagem, de modo a evitar mensagens genéricas e sem personalidade. Usando o nome do modelo como escopo de chaves, você pode personalizar a mensagem sem ter que alterar o template.

Agora vamos para a parte mais legal do capítulo, a criação da sessão do usuário.

9.3 Sessões e cookies

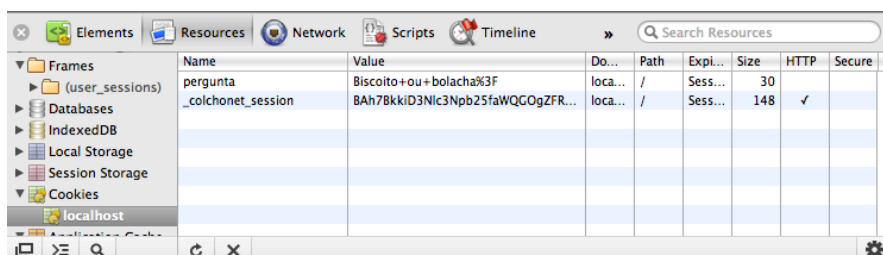
Vamos nos lembrar de uma lição muito importante: **não** é interessante colocar a lógica de negócio nos controles. O que seria lógica de negócio? Nesse caso, é tudo que não é roteamento, mensagem de erro ou decidir a melhor representação para o recurso.

O que vamos fazer no controle é apenas tentar criar a sessão via o modelo `UserSession`. Se algum erro acontecer, simplesmente vamos mostrar ao usuário o template com os erros, e se ocorrer sucesso, direcionaremos o usuário para a página principal com a mensagem apropriada.

Porém, essa ação não será exatamente igual às ações que relacionam-se com o `ActiveRecord`. Os controles possuem os *cookies*, e portanto teremos que passar ao modelo em qual objeto iremos gravar a sessão do usuário (mas não **como** e **quando**, isso é muito importante!).

Nos controles do Rails, temos duas maneiras de acessar os *cookies* dos usuários, via o método `session` e via o método `cookies`. Ambos os métodos se comportam muito parecido com os já famosos `params` e `flash` - na forma de hashes.

Os *cookies* são simples formas de guardar estado entre requisições HTTP. Eles são passados pelo usuário o usuário ao servidor em cada requisição. Por exemplo, se tivermos um controle cuja ação execute o código a seguir, o Rails irá criar uma resposta com o *cookie* informado que, por padrão, irá durar apenas a sessão de navegação do usuário, ou seja, ao fechar o browser, o *cookie* será removido pelo browser.

```ruby
def create
  cookies[:pergunta] = 'Biscoito ou bolacha?'
end
```

O cabeçalho da resposta HTTP será algo parecido com o seguinte. Note, na penúltima linha, o comando para guardar o *cookie* com o que configuramos:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
X-Ua-Compatible: IE=Edge
Etag: "91c95050746190ff27eb34a00497d91b"
Cache-Control: max-age=0, private, must-revalidate
X-Request-Id: 6194eccc0a2d00f4f472d0a79f286dd4
X-Runtime: 0.020849
Content-Length: 2155
Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-04-20)
Date: Tue, 10 Jul 2012 05:57:59 GMT
Connection: Keep-Alive
Set-Cookie: pergunta=Biscoito+ou+bolacha%3F; path=/
Set-Cookie: _colchonet_session=BAh7.....8857; path=/; HttpOnly
```

Por fim, podemos ver no console do browser (Google Chrome, nesse caso) o resultado do *cookie* guardado:

Figura 9.2: Cookie com a chave 'pergunta'

Note também que temos outra chave de *cookie*, a `_colchonet_session`. Essa é gerenciada automaticamente pelo Rails para controlar o CSRF (Cross-Site Request Forgery, visto na seção 5.1).

O nosso objetivo é armazenar o `ID` do usuário autenticado no *cookie*. Portanto, se o email e a senha estiverem corretos e a conta estiver confirmada, salvamos o `ID` do usuário no *cookie* e resgatamos o objeto `User` em cada requisição.

Porém, não é inteligente da nossa parte simplesmente gravar o `ID` do usuário no *cookie* e sentar no sofá esperando a bonança. Isso porquê, ao fazer `cookies[:user_id] = 1`, um usuário mal intencionado pode muito bem pegar o *cookie* local e colocar um outro ID e começar a agir como um outro usuário. Esse ataque é conhecido como *Session Hijacking/Theft* ou sequestro ou roubo de sessão.

O Rails já sabe disso e sabe a melhor maneira de prever ataques como esse. E é justamente essa a grande diferença entre o `cookie` e o `session`; o `session` assinada usando uma chave secreta, de forma que o usuário poderá ler o conteúdo da sessão, porém, ao modificá-la, a assinatura torna-se inválida. Portanto, dificilmente conseguirá forjar uma sessão:

```
BAh7B0kiD3Nlc3Npb25faWQGOgZFRkkiJTBkM2FmNTlkNDcy\
N2VhNDJkZTRhYzE4YjQ0Yjc5OGJkBjsAVEkiEF9jc3JmX3Rva2VuBjsARkki\
MVh1NFMvUWJnOG9pQ0NsbjRkbnducFhTZXEzaUl0NDArVkFQVHVTZi9CZlE9\
BjsARg%3D%3D--100ae84bad4e222ebd17a30f44dac7424549e94f
```

O conteúdo antes do “--” é a sessão do usuário, codificada em Base64. A segunda parte é a assinatura, calculada usando um *token* secreto, que fica no arquivo `config/initializers/secret_token.rb`.

```ruby
# String looooooooonga....
Colchonet::Application.config.secret_key_base = '99289af ...'
```

É muito importante que você nunca divulgue de forma alguma o `secret_key_base` de sua aplicação. Deixar em controle de versão não tem problema (desde que não seja open source!), mas não deixe isso disponível para seus usuários, pois eles podem ter suas contas roubadas!

Se por algum motivo você suspeitar que o `secret_key_base` deixou de ser segredo ou que você tem usuários com sessões roubadas, você pode substituir o código atual gerando um novo via:

```
$ rake secret
513b43f26ce....
```

Isso invalidará todas as sessões e os usuários terão que fazer um novo login, mas é melhor garantir a segurança das informações.

Atualizando sessões para o Rails 4

No Rails 3, as sessões não são encriptadas, apenas assinadas. Isso significa que, mesmo sem a palavra secreta, os usuários podem ver o conteúdo da sessão, só não modificá-la.

No Rails 4, o conteúdo da sessão é encriptada por padrão, que é incompatível com as sessões da versão anterior. Contudo, existe uma forma forma de migrar as sessões dos seus usuários das sessões assinadas para a sessão encriptada de forma transparente para os usuários.

Para fazer a migração, basta que, no arquivo `config/initializers/secret_token.rb`, você configure ambas as opções `secret_key_base`, usada para encriptação (novo), e a `secret_token`, usada para assinatura dos cookies (antigo). Com ambas as chaves presentes, o Rails poderá fazer a migração dos cookies assinados para os cookies encriptados automaticamente.

Por exemplo, se o Colchonet, antes usando Rails 3, possuía o `secret_token.rb` da seguinte forma:

```
ColchoNet::Application.config.secret_token = "123 123 123"
```

No Rails 4, basta especificar também o `secret_key_base`, resultando em:

```
ColchoNet::Application.config.secret_token = "123 123 123"
ColchoNet::Application.config.secret_key_base
    = "ENCRIPTADO ENCRIPTADO"
```

Isso fará com que as sessões dos usuários atuais continuem funcionando e, após o primeiro acesso, o Rails automaticamente irá converter a forma encriptada. Se necessário, use o `rake secret` para gerar o novo valor da chave privada de maneira segura.

Sessões e serialização

É importante lembrar que *cookies*, apesar de não houver limite em sua especificação (RFC 2965), *cookies* possuem uma limitação prática de 4kB, portanto tome cuidado com o que você vai gravar.

Outro problema que pode pegar você de surpresa é a serialização. **Sempre** guarde `IDs` para que objetos sempre estejam em sua versão mais atual quando usados. Há objetos também que não podem ser serializados, como por exemplo, ponteiros a arquivos e objetos de IO, e podem causar exceções na hora de gerar o *cookie*.

Depois de todos esses conceitos sobre *cookies* e sessões, vamos à implementação das sessões de usuário. Para relembrar, devemos usar a `session` para gravar o ID do usuário caso ele seja autenticado com sucesso. Para isso, vamos criar a ação `create` que irá criar uma nova instância da classe `UserSession`. Chamaremos então o método `authenticate!`, que ainda vamos criar. Ele fará o trabalho de gravar na sessão, caso as credenciais estejam corretas. Vamos, por fim, redirecionar o usuário para as páginas correspondentes.

O resultado da ação `create` do controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`) será:

```
class UserSessionsController < ApplicationController
  def new
    @user_session = UserSession.new(session)
  end

  def create
    @user_session = UserSession.new(session, params[:user_session])
    if @user_session.authenticate!
      # Não esqueça de adicionar a chave no i18n!
      redirect_to root_path, notice: t('flash.notice.signed_in')
    else
      render :new
    end
  end

  def destroy
```

```
  end
end
```

Primeiro, precisamos criar o método construtor para aceitar os parâmetros do formulário e a sessão que vem do controle:

```
def initialize(session, attributes={})
  @session = session
  @email = attributes[:email]
  @password = attributes[:password]
end
```

O método `authenticate!` é o método que verifica os dados entrados pelo usuário. Isso será feito através do método `User#authenticate`, que ainda vamos criar. Se tudo estiver correto, ou seja, se existir um usuário cujo email e senhas são válidos, guarda a sessão, caso contrário, marca-se um erro a ser exibido no formulário.

```
def authenticate!
  user = User.authenticate(@email, @password)

  if user.present?
    store(user)
  else
    errors.add(:base, :invalid_login)
    false
  end
end
```

Veja que nesse caso estamos delegando a lógica de verificar se o usuário tem o email e senha válido no modelo `User`. Ou seja, é o modelo usuário que deve saber como a autenticação deve ser feita. É importante centralizar esse tipo de lógica para que, se um dia mudarmos a forma de como o login é feito, basta alterar apenas um lugar.

Usando a API de erros do `ActiveModel` é fácil criar um erro customizado. Nessa situação, o erro não é no atributo `email` e nem no atributo `password`. Por essa razão, adicionamos o erro no `base`, que é o objeto como um todo. Como vimos no template, esse erro será exibido no topo do formulário.

Por fim, temos o método `store`, que grava o ID do usuário na sessão:

```
def store(user)
  @session[:user_id] = user.id
end
```

Por fim, o resultado do modelo `UserSession` (`app/models/user_session.rb`) é:

```
class UserSession
  include ActiveModel::Model

  attr_accessor :email, :password
  validates_presence_of :email, :password

  def initialize(session, attributes={})
    @session = session
    @email = attributes[:email]
    @password = attributes[:password]
  end

  def authenticate!
    user = User.authenticate(@email, @password)

    if user.present?
      store(user)
    else
      errors.add(:base, :invalid_login)
      false
    end
  end

  def store(user)
    @session[:user_id] = user.id
  end
end
```

Vamos agora ao modelo usuário.

9.4 Consultas no banco de dados

Antes de continuar, vamos entender como consultamos o banco de dados através do `ActiveRecord`, buscando no banco de dados sem escrever comandos SQL. É

importante entender que o `ActiveRecord` faz o possível, mas não é infalível. Por isso, é bastante importante entender a chamada gerada e, eventualmente, escrever chamadas SQL manualmente.

Para fazer buscas, o `ActiveRecord` possui uma lista métodos que lembram cláusulas SQL. Por exemplo, o método `limit` irá limitar o número de objetos retornados, sem nenhuma ordem específica. Para fazer buscas filtradas, usamos o método `.where`, e assim por diante. Veja a lista de métodos a seguir:

- `where` - Mapeia cláusulas `WHERE`;
- `select` - Especifica o que será retornado no `SELECT`, ao invés de `*`;
- `group` - Mapeia cláusulas `GROUP BY`;
- `order` - Mapeia cláusulas `ORDER BY`;
- `reorder` - Sobrescreve cláusulas de ordem de `default_scope` (veremos o que é `default_scope` ainda nesse capítulo);
- `reverse_order` - Inverte a ordem especificada (crescente ou decrescente);
- `limit` - Mapeia cláusula `LIMIT`;
- `offset` - Mapeia cláusula `OFFSET`;
- `joins` - Usado para *inner joins* ou quaisquer *outer joins*;
- `includes` - Faz *joins* automaticamente com modelos relacionados, veremos relacionamentos no capítulo 10;
- `lock` - Trava atualizações de objetos para atualização;
- `readonly` - Torna os objetos retornados marcados como apenas leitura;
- `from` - Mapeia cláusula `FROM`;
- `having` - Mapeia cláusula `HAVING`%;

Veja os exemplos abaixo:

```
User.limit 1
#  User Load (0.1ms)  SELECT "users".* FROM "users" LIMIT 1
# => #<ActiveRecord::Relation
#    [#<User id: 11, full_name: "Vinicius Baggio Fuentes", ... >]>

User.where email: 'vinibaggio@example.com'
# User Load (0.1ms) SELECT "users".* FROM "users"
#       WHERE "users"."email" = 'vinibaggio@example.com'

# => #<ActiveRecord::Relation
#    [#<User id: 11, full_name: "Vinicius Baggio Fuentes", ... >]>
```

Escopos

> **Dica: Comandos SQL no console**
>
> Para facilitar o seu aprendizado e também tirar dúvidas, é possível fazer com que o `ActiveRecord` imprima o comando SQL gerado no IRB. Para fazer isso, crie o arquivo `.irbrc` na sua pasta *home* com o seguinte conteúdo:
>
> ```
> if ENV.include?('RAILS_ENV')
>   require 'logger'
>   Rails.logger = Logger.new(STDOUT)
> end
> ```
>
> Lembre-se de reiniciar o console do Rails, `reload!` não é suficiente.

As buscas ficam interessantes quando combinadas Quando chamamos um método de consulta (`.limit` e `.where`, por exemplo), o método retorna um objeto especial chamado `ActiveRecord::Relation`. Esse objeto pode, por sua vez, receber novamente métodos de consulta, acumulando comandos que já foram chamados anteriormente. Veja o exemplo a seguir:

```
User.where(email: 'vinibaggio@example.com').limit(2)
#  User Load (0.3ms)  SELECT "users".* FROM "users"
#          WHERE "users"."email" = 'vinibaggio@example.com'
```

```
#           LIMIT 2
# => <ActiveRecord::Relation [...]>

most_recent = User.order('created_at DESC')

most_recent.limit(1)
# SELECT "users".* FROM "users" ORDER BY created_at DESC LIMIT 1

# Note que a chamada ao .limit anterior não altera
# o objeto most_recent
most_recent.where(email: 'vinibaggio@example.com')
# SELECT "users".* FROM "users"
#  WHERE "users"."email" = 'vinibaggio@example.com'
#  ORDER BY created_at DESC
```

Essa composição de métodos é o que chamamos de **escopo** de busca. Por exemplo, a variável `most_recent` que criamos é um escopo em `User` que sempre irá retornar os usuários em ordem decrescente pela data de criação (`created_at`).

É bom entender como o *query planner* (componente que planeja como será a melhor forma de executar a consulta SQL) do banco de dados vai executar uma consulta. Em banco de dados tradicionais como MySQL e PostgreSQL, é possível observar informações importantes como uso ou não de índices, tabelas temporárias, entre outras coisas, de modo que possamos alterar a consulta para obter melhor performance. Para esses bancos de dados, basta executar `EXPLAIN` na consulta. Com o `ActiveRecord`, conseguimos fazer isso %chamando o método `.explain`:

```
User.where(email: 'vinibaggio@example.com').
     order('created_at DESC').
     explain
# SELECT "users".* FROM "users"
#   WHERE "users"."email" = 'vinibaggio@example.com'
#   ORDER BY created_at DESC

#  EXPLAIN QUERY PLAN SELECT "users".* FROM "users"
#    WHERE "users"."email" = 'vinibaggio@example.com'
#    ORDER BY created_at DESC

# => "EXPLAIN for: SELECT \"users\".* FROM \"users\"
#       WHERE \"users\".\"email\" = 'vinibaggio@example.com'
#       ORDER BY created_at DESC
```

```
#       0|0|0|SEARCH TABLE users
#            USING INDEX index_users_on_email (email=?) (~10 rows)
#       0|0|0|USE TEMP B-TREE FOR ORDER BY"
```

O `ActiveRecord` também permite criar os *named scopes*, ou escopos nomeados, de modo que você chame `.most_recent`, ao invés de `.order('created DESC')`, tornando as chamadas de métodos bastante legíveis. Imagine que a classe `User` tenha o seguinte código:

```ruby
class User < ActiveRecord::Base
  scope :most_recent, -> { order('created_at DESC') }
  scope :from_sampa, -> { where(location: 'São Paulo') }

  # ...
end
```

Encoding

Se você tentar executar esse exemplo e esbarrar com o seguinte problema:

```
SyntaxError: user.rb:3: invalid multibyte char (US-ASCII)
user.rb:3: invalid multibyte char (US-ASCII)
user.rb:3: syntax error, unexpected $end, expecting ')'
  scope :from_sampa, -> { where(:location => 'São Paulo') }
```

Isso deve-se ao fato de que o Ruby não reconheceu esse caractere no código-fonte. Para que o caractere seja reconhecido, coloque o seguinte conteúdo na **primeira** linha do arquivo:

```ruby
# encoding: utf-8
```

> **ATUALIZAÇÃO: ESCOPOS NOMEADOS**
>
> Na versão 3 do Rails, era possível criar escopos nomeados da seguinte forma, sem blocos lambda:
>
> ```
> scope :from_sampa, where(location: 'São Paulo')
> ```
>
> O suporte a essa forma foi abandonado. Você deverá atualizar para a forma com lambdas ou usar métodos de classe diretamente.

Dessa forma, podemos chamar os *named scopes* da mesma maneira que chamamos os outros métodos de busca:

```
User.most_recent.limit(5)
# SELECT "users".* FROM "users"
#   ORDER BY created_at DESC
#   LIMIT 5

User.most_recent.from_sampa
# SELECT "users".* FROM "users"
#   WHERE "users"."location" = 'São Paulo'
#   ORDER BY created_at DESC
```

Ainda é possível criar *named_scopes* com parâmetros:

```
class User < ActiveRecord::Base
  scope :most_recent, -> { order('created_at DESC') }
  scope :from_sampa, -> { where(location: 'São Paulo') }

  scope :from, ->(location) { where(location: location) }

  # ...
end

User.from('San Francisco, CA').most_recent
# SELECT "users".* FROM "users"
#   WHERE "users"."location" = 'San Francisco, CA'
#   ORDER BY created_at DESC

# Se condições se repetirem, apenas a última será mantida:
```

```
User.from('San Francisco, CA').from_sampa
# SELECT "users".* FROM "users"
#   WHERE "users"."location" = 'São Paulo'
```

Por fim, ainda é possível criar um escopo padrão, ou seja, um escopo que será sempre aplicado, via o `default_scope`. Esse escopo não é sobrescrito e se você definir outros `default_scope`, eles são acumulados.

```
class User < ActiveRecord::Base
  default_scope -> { where('confirmed_at IS NOT NULL') }

  # ...
end

User.all
#  SELECT "users".* FROM "users" WHERE (confirmed_at IS NOT NULL)

User.where(location: 'San Francisco, CA')
#  SELECT "users".* FROM "users"
#     WHERE "users"."location" = 'San Francisco, CA'
#     AND (confirmed_at IS NOT NULL)
```

Cuidados com o `default_scope`

O `default_scope` tem utilidades interessantes, como implementar *soft delete*, ou seja, destruir objetos nada mais é do que marcá-lo como destruído e não ser incluso nas buscas. Portanto, um `default_scope` cuja busca filtre esses objetos é uma ótima ideia.

Porém, tome cuidado. Uma das boas práticas da programação sugere que você deve evitar surpresas desagradáveis, e fazer filtros complexos no `default_scope` é uma surpresa infeliz. A razão disso é que o comportamento torna-se implícito e dificilmente o programador que estiver lendo o código (inclusive você mesmo) lembrará da existência de um `default_scope`. Se necessário, peque pelo excesso de clareza ao declarar um escopo nomeado, e não o contrário.

Antes de voltarmos ao projeto, vamos entrar um pouco mais a fundo no `where`, pois ele tem algumas funcionalidades imprescindíveis ao desenvolvedor Rails.

Buscas tradicionais

As buscas mais simples são as buscas que já vimos, para buscar com valores exatos. Para isso, basta passar um hash com a chave sendo a coluna e o valor sendo o valor da busca:

```
User.where(location: 'San Francisco, CA')
# SELECT "users".* FROM "users"
# WHERE "users"."location" = 'San Francisco, CA'
```

Além disso, podemos fazer buscas negadas, usando o `not`:

```
Room.where.not(location: 'San Francisco, CA')
# SELECT "rooms".* FROM "rooms"
# WHERE ("rooms"."location" != 'San Francisco, CA')
```

Buscas manuais

O `ActiveRecord` não possui mapeamento para buscas mais rebuscadas. Nesses casos, precisamos fazer uma busca manualmente. Para isso, basta colocarmos a string com o conteúdo da busca diretamente:

```
User.where('confirmed_at IS NOT NULL')
# SELECT "users".* FROM "users" WHERE (confirmed_at IS NOT NULL)
```

É bastante importante notar que, nesse caso, não usamos nenhuma entrada do usuário. Dessa forma, não precisamos nos preocupar com SQL Injection (ou injeção de SQL). Isso é útil para usar funcionalidades específicas do banco de dados. Porém, raramente é o caso que não precisamos passar um valor para efetuar buscas no banco.

O que é SQL injection?

SQL Injection, ou injeção de SQL, é um ataque a um site de forma que alguém injete códigos SQL maliciosos. Os resultados podem ser drásticos, como conseguir poder de administrador, deletar uma tabela ou roubar dados importantes. Mesmo aparentando ser um ataque simples, ainda é bastante utilizado. A fatídica queda da Playstation Network® em 2011, que afetou o sistema durante meses e milhões de usuários foram prejudicados, foi uma única injeção de SQL. Veja o exemplo a seguir:

```
email = 'vinibaggio@example.com'

User.where("email = '#{email}'")
#  SELECT "users".* FROM "users"
#    WHERE (email = 'vinibaggio@example.com')
#
# => [#<User id: 11, full_name: "Vinicius ... >]

email = "' OR 1=1) --"
User.where("email = '#{email}'")
# SELECT "users".* FROM "users"
#   WHERE (email = '' OR 1=1) --')
#
# => [#<User id: 11, ... >]
```

Esse é um exemplo simples mas prova o ponto que, se você não tomar cuidado com os parâmetros, é possível que um usuário mal intencionado ganhe acesso irrestrito ao seu site.

Buscas parametrizadas via Array

A forma mais simples de busca parametrizada é a forma clássica de consultas que evitam SQL Injection. Para esse tipo de parametrização, usamos o símbolo `?` na consulta e o `ActiveRecord` irá limpar os parâmetros e substituir os símbolos `?`. Para isso, passe uma Array cujo primeiro valor é um "template" da busca e o restante serão os valores a serem substituídos. Exemplo:

```
email = "vinibaggio@example.com"
User.where(["email = ?", email])
```

```
# SELECT "users".* FROM "users"
#   WHERE (email = 'vinibaggio@example.com')
# => [#<User :id: 11, ...>]

email = "' OR 1=1) --"
User.where(["email = ?", email])
# SELECT "users".* FROM "users"
#   WHERE (email = ''' OR 1=1) --')
# => []
```

Buscas parametrizadas via hash

Com muitos parâmetros para substituir, usar o `?` torna o template ilegível. Por isso, é possível usar símbolos no template para substituição de parâmetros. O segundo parâmetro deverá ser um hash que mapeia cada símbolo para um valor:

```
User.where(["location LIKE :location AND email = :email", {
  location: '%CA%',
  email: 'vinibaggio@example.com'
}])

# SELECT "users".* FROM "users"
#   WHERE (
#     location LIKE '%CA%'
#     AND email = 'vinibaggio@example.com'
#   )
```

Buscas em listas e intervalos

Com o `where` ainda é possível fazer buscas em listas e intervalos via o comando SQL `IN` ou `BETWEEN`. Para fazer esse tipo de busca, basta usar a busca tradicional, usando hashes e um Array:

```
User.where(id: [1, 10, 11, 20])
#  SELECT "users".* FROM "users"
#     WHERE "users"."id" IN (1, 10, 11, 20)

User.where(id: 1..10)
# SELECT "users".* FROM "users"
#   WHERE ("users"."id" BETWEEN 1 AND 10)
```

Essas buscas são bastante úteis em datas:

```
User.where(confirmed_at: 1.week.ago..Time.now)
# SELECT "users".* FROM "users"
#   WHERE (
#     "users"."confirmed_at"
#       BETWEEN '2012-07-05 07:37:39.902321'
#       AND '2012-07-12 07:37:39.991792'
#   )
```

9.5 Escopo de usuário confirmado

Agora que temos o conhecimento de como fazer buscas e escopos, implementaremos um método chamado `.authenticate` no modelo usuário. Ele receberá dois parâmetros, email e senha. Precisaremos verificar, em todos os usuários válidos do sistema (ou seja, usuários que confirmaram seu email), qual possui o email e a senha digitados.

Para isso, criaremos um escopo nomeado que retornará todos os usuários que confirmaram sua conta, ou seja, `confirmed_at` é nulo. Adicione o seguinte `scope` no modelo `User` (`app/models/user.rb`):

```
class User
  # ...
  scope :confirmed, -> { where.not(confirmed_at: nil) }
  # ...
end
```

Em seguida, criaremos o método `.authenticate`, que faz a verificação do email e senha. Vamos encadear algumas chamadas: primeiro buscaremos o usuário pelo email. Em seguida, executaremos o método `#authenticate` no usuário retornado. Esse método irá retornar o próprio objeto se a senha estiver correta, caso contrário, retornará `false`:

```
def self.authenticate(email, password)
  user = confirmed.find_by(email: email)
  if user.present?
    user.authenticate(password)
  end
end
```

É possível evitar o `if` usando um método do Ruby chamado `#try`. Nesse método, passamos o nome do método que queremos executar, em seguida os seus parâmetros. Quando o objeto é `nil`, o método não será executado. Se o objeto existir,

o método será executado normalmente, com os parâmetros especificados. Veja os exemplos a seguir:

```
string = "oba"          # => "oba"
string.try(:upcase) # => "OBA"
string = nil            # => nil
string.try(:upcase) # => nil
```

O resultado, aplicando o `try`:

```
def self.authenticate(email, password)
  user = confirmed.
          find_by(email: email).
          try(:authenticate, password)
end
```

Voltemos ao modelo `UserSession` (`app/models/user_session.rb`):

```
def authenticate
  user = User.authenticate(@email, @password)

  if user.present?
    store(user)
  else
    errors.add(:base, :invalid_login)
    false
  end
end
```

Nesse método, que já fizemos, o usuário só será retornado caso o email e senha sejam válidos e o usuário esteja confirmado. Essa parte é muito importante: a classe `UserSession` não precisa saber que existe a lógica de confirmação de usuários, e portanto, delegamos essa lógica apenas ao modelo que deve saber disso.

Com todos os objetos interligados, é possível testar o fluxo completo. Na página http://localhost:3000/user_sessions/new, ao digitar as credenciais corretamente, temos o seguinte resultado:

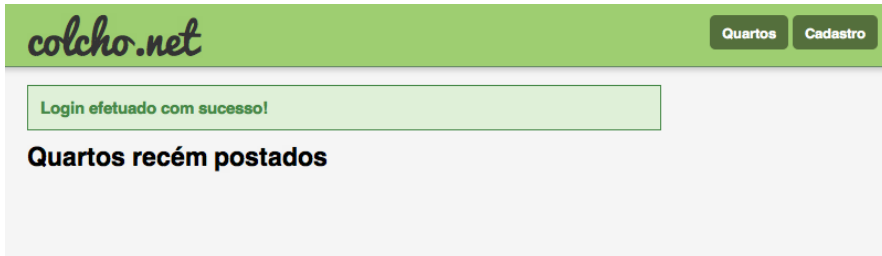



Figura 9.3: Login com credenciais corretas

Agora, quando digitamos as credenciais de forma incorreta:

colcho.net
Quartos
Cadastro
Login
Usuário ou senha inválidos
Email
vinibaggio@example.com
Senha
Entrar
Cadastre-se

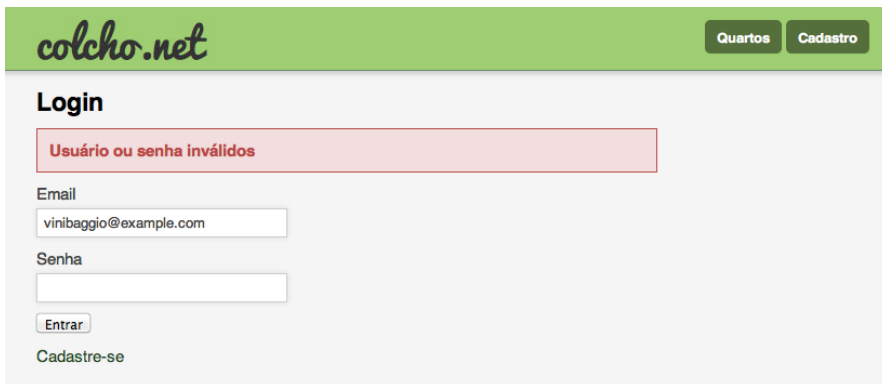



Figura 9.4: Login com credenciais inválidas

Com o login construído, no próximo capítulo vamos melhorar a estrutura da nossa aplicação de forma a mostrar ao usuário autenticado, na barra de navegação, links para editar o perfil e fazer logout. Caso o usuário não estiver logado, mostraremos um link para que ele possa fazer o login.

Vamos também fazer o controle de acesso, como por exemplo, impedir que um usuário edite o perfil de outro usuário, e que um usuário só possa cadastrar um quarto quando estiver autenticado.

Capítulo 10

Controle de acesso

Formação em Ciência da Computação consegue tornar qualquer pessoa um excelente programador tanto quanto estudar pincéis e pigmentos torna qualquer pessoa um excelente pintor.
– Eric Raymond, autor de 'A catedral e o bazar'

Cadastro e login prontos, agora focaremos no controle de acesso. Primeiro, vamos alterar nosso template para exibir uma barra de navegação diferenciada para o usuário logado. Atualmente, a barra é assim:

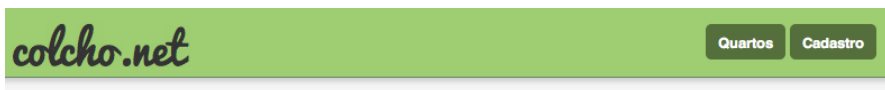



Figura 10.1: Barra de navegação sem informações de sessao

Na versão para o visitante, que não está logado, deverá ficar assim:

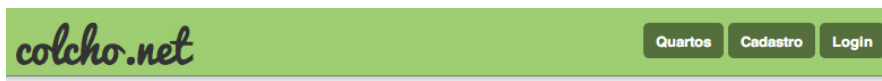



Figura 10.2: Barra de navegação sem login

Por fim, quando o usuário estiver logado, o resultado será:

Figura 10.3: Barra de navegação com perfil do usuário

Em seguida, vamos controlar o fluxo para que um usuário só possa editar o seu próprio perfil. Também vamos corrigir as ações de login e cadastro apenas para quando o usuário **não** estiver logado.

Vamos garantir que, para um usuário cadastrar um quarto, ele precisa estar logado. Junto a isso, vamos associar o quarto cadastrado a ele, aprendendo como criar relacionamento entre modelos `ActiveRecord`. Por fim, vamos também controlar o acesso de edição de quartos, dando a permissão apenas a seus donos, uma lição importante de segurança de dados.

10.1 Helpers de sessão

Antes de alterar a nossa barra, vamos criar métodos para nos auxiliar com a sessão do usuário. Como esse conceito é inerente à aplicação inteira, vamos criar os seguintes métodos no `ApplicationController`:

- `user_signed_in?` - Método que verifica se o usuário possui uma sessão autenticada;
- `current_user` - Retorna o objeto da classe `User` que está na sessão atual;
- `require_authentication` - Filtro que força a autenticação de usuários, redirecionando-o à pagina de login caso não esteja logado, e não fazendo nada caso o usuário esteja com uma sessão válida;

- `require_no_authentication` - Filtro para evitar que usuários cadastrados tentem acessar páginas que só deverão ser acessadas quando o usuário não tiver um login (como páginas de cadastro e de login).

Vamos lá! Para criar essa lógica, vamos ter que adicionar um pouco de comportamento na classe de sessões de usuário, a `UserSession`. Criaremos o método que buscará o usuário na sessão (`current_user`). Precisamos saber também se a sessão do usuário é válida, portanto verificaremos a presença do ID do usuário na sessão, no método (`user_signed_in?`):

```
class UserSession
  #...
  def current_user
    User.find(@session[:user_id])
  end

  def user_signed_in?
    @session[:user_id].present?
  end
end
```

Com esses dois métodos no `UserSession`, conseguimos fazer tudo o que precisamos no `ApplicationController`. Lembre-se que, ao criar um método no `ApplicationController`, todas os controles da aplicação que herdam dela também terão esses métodos. Adicionaremos então métodos para obter as informações sobre a sessão e então, criaremos os métodos que serão usados como filtros de requisições.

Em primeiro lugar, criaremos o método `user_session`, que criará uma instância do `UserSession` baseada na sessão do usuário. Os outros devem chamar comportamentos que já implementamos na classe `UserSession`, portanto, usaremos o método `delegate`. O `delegate`, extensão do `ActiveSupport`, criará métodos na classe que repassarão a mensagem para o objeto mencionado. A princípio parece ser trabalhoso, mas é importante lembrar das responsabilidades de cada componente.

```
class ApplicationController < ActionController::Base
  delegate :current_user, :user_signed_in?, to: :user_session

  # ...
```

```
  def user_session
    UserSession.new(session)
  end
end
```

Os filtros serão dois: o `require_authentication`, que irá redirecionar o usuário à pagina de login caso a sessão não seja válida, com uma mensagem. Em seguida, o `require_no_authentication` irá redirecionar o usuário para a página principal, com uma mensagem.

```
class ApplicationController < ActionController::Base
  # ...

  def require_authentication
    unless user_signed_in?
      redirect_to new_user_sessions_path,
        alert: t('flash.alert.needs_login')
    end
  end

  def require_no_authentication
    if user_signed_in?
      redirect_to root_path,
        notice: t('flash.notice.already_logged_in')
    end
  end
end
```

É importante ressaltar um comportamento de filtros: se o filtro executar uma redireção ou alguma renderização de template (via `redirect_to` ou `render`), a ação e filtros seguintes **não** serão executados. Isso é bastante conveniente para os filtros `require_authentication` e `require_no_authentication`, mas é importante lembrar desse fato em filtros que você criar no futuro. Isso é *quase* tudo que precisamos.

Os métodos `current_user` e `user_signed_in?` podem ser usados em controles, mas é importante usá-los também nos templates. Para disponibilizar um método do controle nos templates, usamos a *class macro* `helper_method`. Basta fazer:

```
class ApplicationController < ActionController::Base
  delegate :current_user, :user_signed_in?, to: :user_session
```

```
  helper_method :current_user, :user_signed_in?

  # ...
end
```

Templates da barra

Agora que temos os métodos de controle de sessão do usuário, já é possível aprimorar nossa barra de navegação. Para isso, vamos ao layout da aplicação (app/views/layouts/application.html.erb) e vamos extrair a barra de navegação em uma *partial*:

```
...
<header>
  <div id="header-wrap">
    <h1><%= link_to "colcho.net", root_path %></h1>
    <% if user_signed_in? %>
      <%= render 'layouts/user_navbar' %>
    <% else %>
      <%= render 'layouts/visitor_navbar' %>
    <% end %>
  </div>
</header>
...
```

Agora vamos criar o template app/views/layouts/_visitor_navbar.html.erb. Note que tivemos que especificar o caminho da *partial*. Isso deve-se ao fato de que, como o layout é executado em todas as ações, o caminho de busca de templates fica relativo a cada controle. Especificando o caminho, garantimos que, não importa o controle em execução, a *partial* sempre será achada. Temos que fazer o mesmo com as chaves de tradução:

```
<nav>
  <ul>
    <li><%= link_to t('layout.rooms'), rooms_path %></li>
    <li><%= link_to t('layout.signup'), new_user_path %></li>
    <li><%= link_to t('layout.signin'), new_user_sessions_path %></li>
  </ul>
</nav>
```

O resultado é o seguinte:

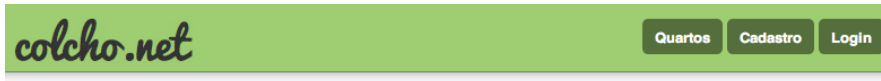



Figura 10.4: Barra de navegação sem login

A barra de navegação para usuários logados fica da seguinte forma (`app/views/layouts/_user_navbar.html.erb`):

```
<nav>
  <ul>
    <li><%= link_to t('layout.new_room'), new_room_path %></li>
    <li><%= link_to t('layout.rooms'), rooms_path %></li>
    <li>
      <%= link_to t('layout.my_profile'),
                  user_path(current_user) %>
    </li>
    <li>
      <%= link_to t('layout.signout'), user_sessions_path,
                   method: :delete %>
    </li>
  </ul>
</nav>
```

Para fazer o logout, o que precisamos é apagar a sessão do usuário, via a ação `destroy`. De acordo com a nossa rota, para executar essa ação, precisamos fazer `DELETE /user_sessions`. O problema é que essa ação HTTP não é suportada por todos os browsers, portanto o Rails tem um mecanismo para executar esse tipo de operação:

```
<a href="/pt-BR/user_sessions" data-method="delete"
                                      rel="nofollow">
  Logout
</a>
```

O código anterior possui um atributo `data-*`, válido no HTML 5. Ele é usado para colocar, em templates HTML, dados que possam ser usados de outra forma, como por exemplo, JavaScript.

O Rails então usa o `data-method` via JavaScript para simular o `DELETE`. A forma que isso é feito é através da criação de um formulário (usando `POST`), e in-

cluindo um campo chamado `_method`. No processamento da requisição, esse atributo faz com que o Rails interprete a requisição como `DELETE`. Com outros tipos de clientes HTTP `browsers`, é possível chamar o `DELETE` diretamente.

Por fim, adicionamos novas chaves na seção "layout" do arquivo de I18n (`config/locales/pt-BR.yml`):

```
layout:
  rooms: Quartos
  new_room: Cadastre seu quarto!
  signup: Cadastro
  signin: Login
  my_profile: Meu perfil
  signout: Logout
```

O resultado é:

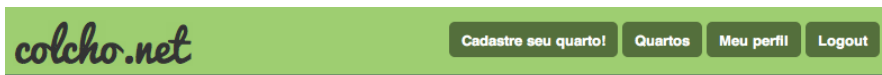



Figura 10.5: Barra de navegação com perfil do usuário

Todos os links estão funcionais e traduzidos, porém ainda não implementamos a ação de destruir a sessão do usuário, e é isso que vamos fazer agora.

Logout

Vamos primeiro à classe `UserSession` (`app/models/user_session.rb`). Para removermos um item da sessão, basta associar `nil` ao item:

```
class UserSession
  # ...
  def destroy
    @session[:user_id] = nil
  end
end
```

Isso é suficiente para que todo o login seja desfeito, ou seja, o *cookie* será alterado e todo o resto da aplicação não detectará o usuário como logado. Agora precisamos chamar esse método no controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`):

```
class UserSessionsController < ApplicationController
  # ...

  def destroy
    user_session.destroy
    redirect_to root_path, notice: t('flash.notice.signed_out')
  end
end
```

O método `user_session` foi criado no `ApplicationController` e vamos usá-lo para tornar o uso desse objeto mais fácil. Basta agora incluir essa mensagem no I18n (`config/locales/pt-BR.yml`) e acabamos a barra!

```
flash:
  alert:
    needs_login: "Você precisa estar logado para continuar"
  notice:
    signed_in: 'Login efetuado com sucesso!'
    signed_out: 'Logout efetuado com sucesso. Até logo!'
    already_logged_in: "Você já está logado."
```

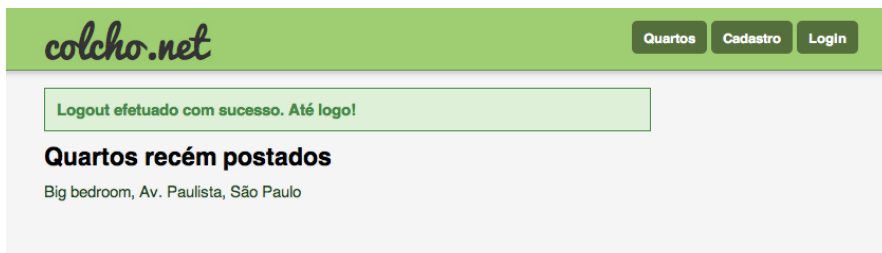



Figura 10.6: Mensagem de logout após clicar no link

10.2 Não permita edição do perfil alheio

Depois do cadastro e do login, ainda é possível editar qualquer perfil. Para ver isso funcionando, basta ir ao seu perfil e mudar o `ID` na rota para outro usuário que você tenha cadastrado.

Para prevenir essa situação, vamos usar um filtro: verificamos se o usuário está logado e se o usuário logado é o mesmo que tentamos editar ou atualizar. Basta então

comparar o ID do usuário na rota com o usuário da sessão. Caso não sejam iguais, redirecionamos o usuário apenas para a visualização do perfil. Aplicaremos o filtro apenas nas ações edit e update.

O filtro can_change, no controle de usuários (app/controllers/users_controller.rb), ficará assim:

```
class UsersController < ApplicationController
  before_action :can_change, only: [:edit, :update]

  # ...

  private

  def user_params
    params.
      require(:user).
      permit(:email, :full_name, :location, :password,
             :password_confirmation, :bio)
  end

  def can_change
    unless user_signed_in? && current_user == user
      redirect_to user_path(params[:id])
    end
  end

  def user
    @user ||= User.find(params[:id])
  end
end
```

Usamos no método user uma expressão idiomática de Ruby chamada *memoization* (ou "memorização"). Nessa expressão, apenas associamos a variável (e por consequência, fazemos a busca no banco de dados) caso ela nunca tenha sido iniciada. Você pode pensar nisso como um cache de variável.

Aproveitando que estamos no controle de usuários, vamos forçar que, para a página de cadastro, é necessário não estar logado. Para isso, basta usar o filtro que criamos, require_no_authentication. Mesmo que não haja acesso através das páginas, é uma boa ideia proteger essas ações:

```
class UsersController < ApplicationController
  before_action :require_no_authentication, only: [:new, :create]
  before_action :can_change, only: [:edit, :update]

  #...
end
```

Adicionaremos o mesmo filtro na ação de login (tanto para o formulário quanto para a ação `create`), no controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`). Filtraremos também a ação `destroy` para apenas usuários logados (com o único intuito de evitar ver o `flash` mesmo não tendo feito ação alguma):

```
class UserSessionsController < ApplicationController
  before_action :require_no_authentication, only: [:new, :create]
  before_action :require_authentication, only: :destroy

  # ...
end
```

Criaremos esses filtros para o cadastro e atualização de quartos (aproveitamos para fazer uma limpeza, tal como colocar I18n, remover os comentários e remover o tratamento para respostas em JSON), no controle `RoomsController` (`app/controllers/rooms_controller.rb`):

```
class RoomsController < ApplicationController
  before_action :require_authentication,
    only: [:new, :edit, :create, :update, :destroy]

  def index
    @rooms = Room.all
  end

  def show
    @room = Room.find(params[:id])
  end

  def new
    @room = Room.new
  end
```

```
  def edit
    @room = Room.find(params[:id])
  end

  def create
    @room = Room.new(params[:room])

    if @room.save
      redirect_to @room, notice: t('flash.notice.room_created')
    else
      render action: "new"
    end
  end

  def update
    @room = Room.find(params[:id])

    if @room.update(params[:room])
      redirect_to @room, notice: t('flash.notice.room_updated')
    else
      render action: "edit"
    end
  end

  def destroy
    @room = Room.find(params[:id])
    @room.destroy

    redirect_to rooms_url
  end
end
```

Por que declarar todas as ações nos filtros?

Para todas as opções de segurança, prefira ser específico. No caso anterior, poderíamos fazer:

```
before_action :require_authentication,
  except: [:index, :show]
```

Porém, novas ações automaticamente terão o filtro `require_authentication` aplicado e não necessariamente isso é o que nós queremos. Além disso, ao ser específico, é mais fácil perceber o que está acontecendo e ajuda a nos lembrar a configurar as permissões corretamente.

Concluímos a parte de filtros, fazendo com que o usuário tenha que estar logado (ou não) em algumas situações. Ainda temos que fazer o controle de permissões, ou seja, não permitir que um usuário não possa editar um quarto que não pertença a ele. Para que possamos adicionar posse de objetos no sistema, é necessário criar relacionamentos. Relacionamentos entre objetos no `ActiveRecord` é bastante fácil de usar, e é isso que vamos estudar em seguida.

10.3 Relacionando seus objetos

Em sistemas envolvendo bancos de dados, é necessário identificar um objeto univocamente. A estratégia mais comum para gerar essa identificação é via a geração de números sequenciais. Essa estratégia é tão comum que a maioria dos bancos de dados possuem uma implementação embutida. Dessa maneira, ao anotar o campo como um identificador, ou **chave primária**, toda vez que criarmos um objeto, o banco de dados poderá, automaticamente, incrementar o valor do contador e associá-lo ao novo registro.

É possível armazenar chaves primárias em outras tabelas, de modo a criar uma espécie de ponteiro. Esse campo é conhecido como chave estrangeira, ou pelo nome em inglês, *foreign key*. Com as chaves estrangeiras, é possível implementar diversos tipos de relacionamentos.

O relacionamento mais simples é conhecido como o relacionamento um-para-um, ou seja, um objeto possui uma referência direta a outro. Um exemplo seria uma

conta de usuário, na tabela `profiles`, que possuiria uma foto de perfil, da tabela `profile_pictures`. Na tabela `profile_pictures` criamos registros que possuam a chave estrangeira `user_id`, apontando para o usuário que é dono da foto. É comum também referirmos a esse relacionamento como `profile_pictures` **pertence a** (*belongs to*) `users`.

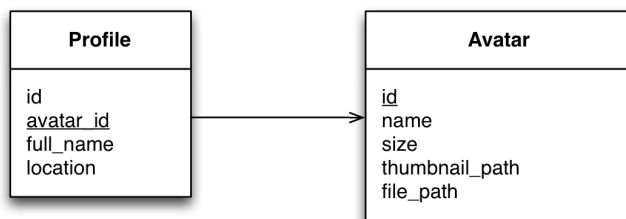



Figura 10.7: Diagrama do relacionamento um-para-um

A modelagem um-para-muitos é a modelagem mais popular. Nela, um objeto pode possuir nenhum, um, ou muitos outros objetos relacionados. Um exemplo é o que teremos no próprio colcho.net: um usuário pode ter nenhum quarto, um quarto apenas ou o número que quiser cadastrar, sem restrições. A implementação desse relacionamento é da mesma maneira que o relacionamento um-para-um, porém não há imposições lógicas de quantos elementos associados podem existir.

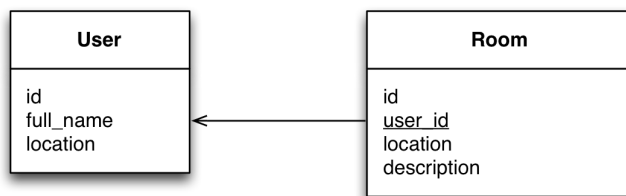



Figura 10.8: Diagrama do relacionamento um-para-muitos

Por fim, o último tipo de relacionamento que o Rails nos ajuda a fazer é o relacionamento muitos-para-muitos. Para essa associação existir, é necessário uma tabela intermediária, chamada tabela de ligação. Imagine uma situação em que um usuá-

rio possa participar de vários projetos e que um projeto possua vários membros. É necessário criar uma tabela que contenha duas chaves-estrangeiras, uma apontando ao usuário e outra apontando ao projeto.

É natural que as tabelas de ligação acabem ganhando vida própria como um conceito dentro do sistema, ou seja, elas possuam outros atributos além de apenas chaves-estrangeiras. Para o caso do exemplo anterior, é natural chamar a tabela de ligação de "participação".

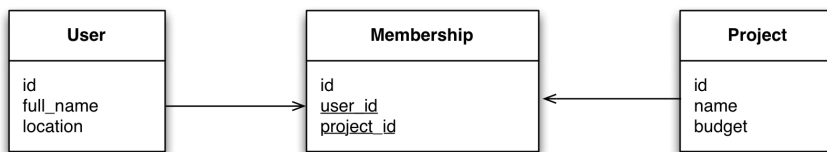



Figura 10.9: Diagrama do relacionamento muitos-para-muitos

O `ActiveRecord` possui facilidades para os três tipos de relacionamentos que vimos. Vamos aprender na prática como vamos criar uma associação um-para-muitos e muitos-para-muitos. Deixaremos a implementação de relacionamentos um-para-um de lado pois a implementação é praticamente a mesma dos relacionamentos muitos-para-muitos.

10.4 Relacione quartos a usuários

O que vamos fazer é criar a associação um-para-muitos entre usuários e quartos. Um usuário poderá ter nenhum, um, ou vários quartos. Para isso, precisamos adicionar uma chave-estrangeira no modelo `Room` para apontar para `User`. Criamos então uma migração:

```
$ rails g migration add_user_id_to_rooms user:references
  invoke  active_record
  create    db/migrate/20130719075333_add_user_id_to_rooms.rb
```

Como usamos `references` como tipo da coluna, o Rails já gerou o código apropriado para criar a chave-estrangeira, adicionando uma referência (no caso do Rails, um campo de números inteiros) que possui um índice, para tornar o acesso mais rápido.

```ruby
class AddUserIdToRooms < ActiveRecord::Migration
  def change
    add_reference :rooms, :user, index: true
  end
end
```

Convenção sobre chaves-estrangeiras

A melhor maneira é seguir os padrões do Rails e usar os métodos que ele provê para você. Assim você poderá usar o máximo de funcionalidades que o Rails possui, sem a necessidade de ter que reimplementar o que o Rails já faz.

Caso não seja possível fazer como fizemos anteriormente, é possível ainda garantir o funcionamento de relacionamentos com o `ActiveRecord`. Basta que você nomeie sua chave-estrangeira como `nome_do_relacionamento` + `_id`, nesse caso, `user_id`. Fazendo dessa maneira, o `ActiveRecord` é capaz de derivar automaticamente o nome da coluna para fazer uniões de tabelas.

Note que o `ActiveRecord` não cria nenhum mecanismo de chaves-estrangeiras no banco de dados (como *constraints*), portanto se você tiver interesse em fazer isso, deverá executar SQL manualmente, usando o método `execute`, lembrando que isso resultará no acoplamento de suas migrações em um banco de dados específico.

Detalhes de banco de dados

É recomendado que você estude o sistema que está usando com cuidado (como toda ferramenta!) e aplicar configurações e boas práticas de uso. Por exemplo, se seu sistema de banco de dados suporta *constraints* de chaves-estrangeiras, tal como o PostgreSQL, não tenha medo de usá-lo só porque o Rails não possui um método para tal.

Após executar `rake db:migrate`, editaremos o modelo `Room` (`app/models/room.rb`) e colocar uma *class macro* para indicar que o modelo quarto pertence a um usuário.

Com a *class macro* `belongs_to`, é possível associar os objetos na relação quarto `->` usuário. O `ActiveRecord` já sabe qual objeto deve criar devido ao nome do relacionamento (`:user`) e também já sabe qual campo usar para buscar o objeto devido (`user_id`):

```
class Room < ActiveRecord::Base
  belongs_to :user

  # ...
end
```

Veja o seguinte exemplo:

```
user = User.first
# => #<User id: 11, ... >

room = Room.first
# => #<Room id: 3, ..., user_id: nil>

room.user = user
# => #<User id: 11, ... >

room.save
# => true

room.user
# => #<User id: 11, ...>
```

Com esse relacionamento, já é possível mostrar, na página de um quarto, o seu dono. Mas antes de chegar lá, ainda não é possível saber quais quartos um usuário possui. Seguindo todas as convenções do Rails, adicionar a outra parte do relacionamento é apenas chamar a *class macro* `has_many`.

A *class macro* `has_many` faz várias coisas para descobrir o relacionamento. Primeiro, como é um relacionamento um-para-muitos (*has many* significa, literalmente, "tem muitos"), o nome do relacionamento deverá estar no plural e portanto, o modelo é o `Room`. Dada a natureza do relacionamento, o `ActiveRecord` sabe também que o modelo `room` deverá ter um campo para o próprio modelo `user` (`user_id`) e então finalmente, consegue buscar todos os quartos que pertencem a um usuário.

Basta então adicionar uma única linha no modelo `User` (`app/models/user.rb`):

```
class User < ActiveRecord::Base
  EMAIL_REGEXP = /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/

  has_many :rooms

  # ...
end
```

Veja o exemplo abaixo:

```
user = User.first
# => #<User id: 11, ... >
user.rooms
# => #<ActiveRecord::Associations::CollectionProxy
# [#<Room id: 2, ... ]>
```

Note no exemplo anterior como o relacionamento `rooms` retorna algo *parecido* com um `Array`. Na verdade, esse objeto nada mais é que um escopo. Isso significa que ainda é possível aplicar qualquer método de busca que vimos na seção "9.4 Consultas no banco de dados", inclusive escopos nomeados:

```
user.rooms.where('title like ?', '%big%')
#  SELECT "rooms".* FROM "rooms"
#     WHERE "rooms"."user_id" = 11 AND (title like '%big%')
#
# => #<ActiveRecord::Relation [#<Room id: 3, title: "Big bedroom", ...>]>
```

Pronto, o relacionamento um-para-muitos está criado! Veremos no capítulo 11, "Avaliação de quartos, relacionamentos muitos para muitos e organização do código" como criar o tipo de relacionamento muitos-para-muitos.

E o relacionamento um-para-um?

O relacionamento um-para-um é quase igual ao relacionamento um-para-muitos. A diferença é que, ao invés de usar a *class macro* `has_many`, você usa o `has_one` (deixando o nome no singular, como `user` ao invés de `users`). É importante que o `belongs_to` **sempre** fique no modelo que possui a chave-estrangeira.

Relacionamentos prontos, vamos dar início ao tratamento de segurança de dados.

10.5 Limite o acesso usando relacionamentos

Os escopos são ótimos para controlar permissões de acesso e permitir apenas que dados interessantes sejam exibidos. Como os relacionamentos também são, é possível usá-los para limitar o que o usuário pode alterar.

Para isso, vamos usar a seguinte técnica: ao invés de fazer a busca (com o `.find`) do modelo via seu `id`, sempre que formos criar, atualizar ou deletar um objeto do banco, vamos limitar o escopo de acesso ao usuário logado. Dessa forma, se o usuário logado não tiver permissão para acessar aquele objeto, o usuário irá deparar-se com um erro de Página não encontrada (404).

Usar a associação possui outra vantagem: construir novos objetos usando a associação fará com que o objeto criado já tenha a sua chave-estrangeira preenchida corretamente. Veja o exemplo:

```
user = User.first
# => #<User id: 11, ... >
user.rooms.build title: 'Quarto aconchegante',
                 description: 'Quarto grande com muita luz natural.'
# => #<Room id: nil,
#         title: "Quarto aconchegante",
#      location: nil,
#   description: "Quarto grande com bastante luz natural.",
#    created_at: nil,
#    updated_at: nil,
#       user_id: 11>

room.user
# => #<User id: 11, ...>
```

Isso é útil para não termos que associar nos preocupar em ligar os objetos. Vamos aplicar essa ideia no controle `RoomsController`, mas antes vamos fazer uma pausa para uma lição importante: segurança de dados.

O Diaspora (http://joindiaspora.com) é uma alternativa livre ao Facebook (http://facebook.com) . Sua proposta é que, ao contrário do Facebook, os dados dos usuários são **realmente** privados e a plataforma possui código-fonte livre para ser investigado. Você pode, por exemplo, instalar o Diaspora em um servidor privado seu e possuir sua própria rede social. Por causa de sua causa nobre, fez bastante barulho nos Estados Unidos no seu lançamento, porém, também foi um grande fiasco técnico no lançamento.

A plataforma que deveria ser segura e privada possuía grandes falhas de segurança de forma que qualquer usuário poderia ver, criar e editar fotos e outros conteúdos de qualquer usuário. Obviamente os desenvolvedores do Diaspora aprenderam com os erros e já melhoraram a plataforma. Apesar de uma falha simples, muitas aplicações Rails possuem o mesmo problema.

O problema estava nos controles, e temos exatamente a mesma situação ocorrendo no controle do recurso quarto: como só buscamos o objeto por `id`, podemos forjar uma requisição de alteração apenas alterando a URL. Com isso, podemos acessar e alterar recursos que eu, como usuário, não deveria ter acesso. Veja o nosso controle `RoomsController`, no qual buscamos o quarto pelo parâmetro na URL:

```
class RoomsController < ApplicationController
  before_action :require_authentication,
    only: [:new, :edit, :create, :update, :destroy]

  # ...
  def update
    @room = Room.find(params[:id])

    if @room.update(params[:room])
      redirect_to @room, notice: t('flash.notice.room_updated')
    else
      render action: "edit"
    end
  end
end
```

O que acontece é que temos o filtro para impedir que um usuário não logado faça atualizações (e outras ações também, como remover) em **qualquer** objeto, mas isso não implica que um usuário não pode atualizar um quarto que não pertence a ele.

Para explorar essa falha de segurança, basta abrir um formulário de edição de um quarto que pertença a você, alterar o ID do quarto na URL, e submeter o formulário. O código irá buscar pelo quarto, que existe e é valido, e irá atualizar o objeto.

A melhor forma de impedir isso é **sempre** usar escopos quando fizermos buscas de objetos. No nosso caso, o usuário atual possui quartos, então vamos usar o `has_many` a nosso favor:

```
1 class RoomsController < ApplicationController
2   before_action :require_authentication,
```

```
3     only: [:new, :edit, :create, :update, :destroy]
4
5   # ...
6
7   def update
8     @room = current_user.rooms.find(params[:id])
9
10    if @room.update(room_params)
11      redirect_to @room, notice: t('flash.notice.room_updated')
12    else
13      render action: "edit"
14    end
15  end
16
17  #...
18
19  private
20
21  def room_params
22    params.
23      require(:room).
24      permit(:title, :location, :descriptions)
25  end
26 end
```

A única alteração foi a linha 8, ao invés de buscarmos no modelo, buscamos pelos quartos que o usuário tem acesso. Dessa forma, mesmo se um usuário mal intencionado alterar o ID do recurso no formulário, a atualização do modelo não irá ocorrer. Isso acontece porque, ao tentar buscar em seus quartos, o objeto não será encontrado. O `ActiveRecord` então irá disparar a exceção `ActiveRecord::RecordNotFound` e o controle irá retornar um erro `404 Not found` (não encontrado) para o usuário, o comportamento correto.

Aplicando essa correção segurança (e aproveitando para fazer limpezas no código), o controle `RoomsController` (`app/controllers/rooms_controller.rb`) fica, por fim:

```
class RoomsController < ApplicationController
  before_action :require_authentication,
    only: [:new, :edit, :create, :update, :destroy]
```

```
def index
  # Exercício pra você! Crie um escopo para ordenar
  # os quartos dos mais recentes aos mais antigos.
  @rooms = Room.most_recent
end

def show
  @room = Room.find(params[:id])
end

def new
  @room = current_user.rooms.build
end

def edit
  @room = current_user.rooms.find(params[:id])
end

def create
  @room = current_user.rooms.build(room_params)

  if @room.save
    redirect_to @room, notice: t('flash.notice.room_created')
  else
    render action: "new"
  end
end

def update
  @room = current_user.rooms.find(params[:id])

  if @room.update(room_params)
    redirect_to @room, notice: t('flash.notice.room_updated')
  else
    render action: "edit"
  end
end

def destroy
  @room = current_user.rooms.find(params[:id])
  @room.destroy
```

```
    redirect_to rooms_url
  end

  private

  def room_params
    params.
      require(:room).
      permit(:title, :location, :descriptions)
  end
end
```

> **Quartos já existentes**
>
> Se você já tem algum quarto cadastrado antes dessas alterações, você não vai mais conseguir editá-los ou removê-los. O jeito é destruir todos os quartos (fazendo `Room.destroy_all` no console) ou atualizar todos de modo que você seja o dono (`Room.update_all :user_id => User.first.id`).

Com essa modificações, garantimos a segurança dos dados de forma simples e legível. Agora vamos melhorar os templates para não exibir os links de remoção e edição caso o usuário não seja o dono e vamos aproveitar para melhorar o visual em geral.

10.6 Exibição e listagem de quartos

Por enquanto, temos a listagem da seguinte forma:

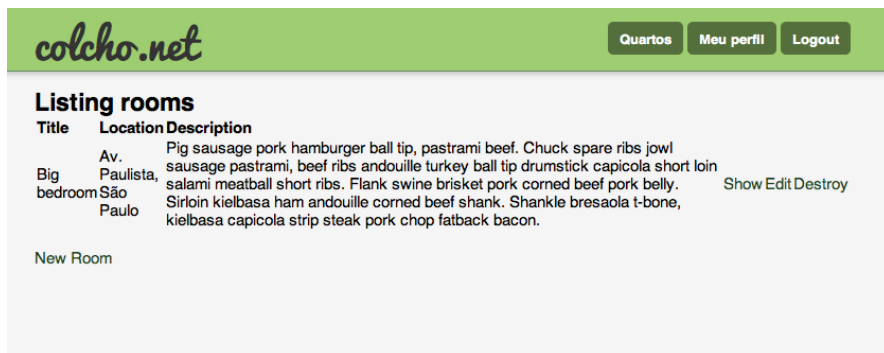



Figura 10.10: Listagem de quartos sem nenhum tratamento

Isso ainda pode melhorar. Embora o *scaffold* seja bom para começar e ter uma ideia do que queremos, o resultado dele sempre precisa ser trabalhado. O objetivo é tornar a listagem mais elegante:

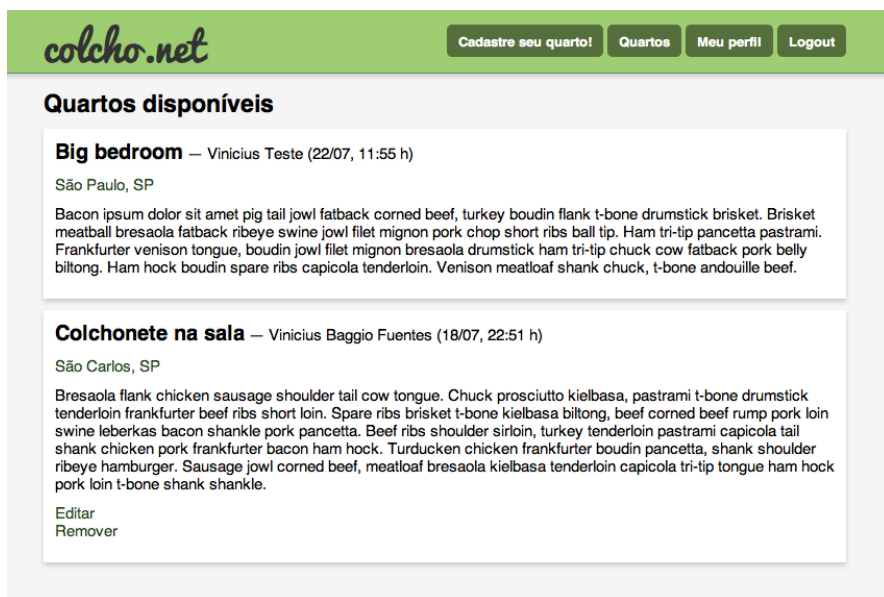



Figura 10.11: Listagem de quartos com estilo

Antes de criar o template, vamos precisar de um `helper`. Esse `helper` verificará se o usuário está logado e se o quarto pertence a ele. Vamos usar ele para exibir

os links de edição e remoção do modelo apenas aos donos. Abra o `RoomsHelper` (`app/helpers/rooms_helper`) e crie o método `belongs_to_user` como a seguir:

```
module RoomsHelper
  def belongs_to_user(room)
    user_signed_in? && room.user == current_user
  end
end
```

Em seguida, vamos alterar o template da ação `index` (`app/views/rooms/index.html.erb`), para apenas exibir os links de edição e remoção caso o modelo pertença ao usuário logado, além de outros detalhes como link para o Google Maps e i18n:

```
<h1><%= t '.title' %></h1>

<% @rooms.each do |room| %>
  <article class="room">
    <h2><%= link_to room.title, room %></h2>
    <span class="created">
      <%= t '.owner_html',
        owner: room.user.full_name,
        when: l(room.created_at, format: :short) %>
    </span>
    <p>
      <span class="location">
        <%= link_to room.location,
                    "https://maps.google.com/?q=#{room.location}",
                    target: :blank %>
      </span>
    </p>
    <p><%= room.description %></p>
    <% if belongs_to_user(room) %>
      <ul>
        <li><%= link_to t('.edit'), edit_room_path(room) %></li>
        <li><%= link_to t('.destroy'), room_path(room),
                  method: :delete, data:
                  {confirm: t('dialogs.destroy')}
            %></li>
      </ul>
```

```
    <% end %>
  </article>
<% end %>
```

No template anterior temos uma novidade: o uso do `l`. O `l`, atalho para `localize`, faz parte do sistema de internacionalização, mas com um papel diferente. O `l` faz a "tradução" de datas e horário, usando o formato adequado para cada idioma. Por exemplo, no Brasil usamos datas no formato dia, mês e ano. Porém, nos Estados Unidos, o mais comum é usar mês, dia e ano. Uma vez tendo o arquivo de suporte do idioma, o I18n faz essa "tradução" automaticamente.

O `localize` ainda aceita alguns formatos de data, tal como `:short`, `:long` e o padrão (`:default`, no YAML). Você pode ainda criar os formatos que quiser, seguindo o padrão `strftime` (padrão de formatação de hora e data de sistemas POSIX). Veja os formatos que já vem com o Rails:

```
formats:
  default: ! '%A, %d de %B de %Y, %H:%M h'
  long: ! '%A, %d de %B de %Y, %H:%M h'
  short: ! '%d/%m, %H:%M h'
```

Para saber o que é cada símbolo, você pode consultar a documentação do Ruby no método `strftime`, ou digitar `man strftime`, caso você esteja em OS X ou Linux.

Partials de modelos

Uma convenção importante do Rails são as *partials* de modelos. Veja o exemplo a seguir:

```
<%= render @room %>
```

Se o objeto `@room` for uma instância do modelo `Room`, o Rails irá buscar pela *partial* `_room.html.erb`, e é exatamente o que vamos fazer para deixar o template mais limpo:

```
<h1><%= t '.title' %></h1>

<% @rooms.each do |room| %>
  <%= render room %>
<% end %>
```

Além disso, o Rails é capaz de renderizar coleções, ou seja, se você passar um `Array` ou relacionamentos do `ActiveRecord`, por exemplo, o Rails irá renderizar cada elemento da lista. Com ambas as alterações, a listagem de quartos (`app/views/rooms/index.html.erb`) fica da seguinte maneira:

```
<h1><%= t '.title' %></h1>

<%= render @rooms %>
```

E a *partial* (`app/views/rooms/_room.html.erb`):

```
<article class="room">
  <h2><%= link_to room.title, room %></h2>
  <span class="created">
    <%= t '.owner_html',
      owner: room.user.full_name,
      when: l(room.created_at, format: :short) %>
  </span>
  <p>
    <span class="location">
      <%= link_to room.location,
                  "https://maps.google.com/?q=#{room.location}",
                  target: :blank %>
    </span>
  </p>
  <p><%= room.description %></p>
  <% if belongs_to_user(room) %>
    <ul>
      <li><%= link_to t('.edit'), edit_room_path(room) %></li>
      <li><%= link_to t('.destroy'), room_path(room),
                method: :delete, data:
                {confirm: t('dialogs.destroy')}
          %></li>
    </ul>
  <% end %>
</article>
```

Note que na *partial* fazemos referência ao quarto usando a variável `room`. Isso acontece porque, quando usamos `render` em um objeto, o Rails mapeia o nome do objeto a ser renderizado pelo nome da *partial*. Tome cuidado, pois isso é implícito e fica difícil de descobrir o que está acontecendo em algumas situações. Se você preferir, podemos ser explícito em qual objeto devemos mapear para a *partial*

```
<%= render partial: 'room', object: current_user.rooms.first %>
```

O nome do objeto na *partial* ainda será `room`, por causa do nome da *partial*, mas o objeto que estamos renderizando é o `current_user.rooms.first`.

Por fim, ainda é possível especificar o nome do objeto através do uso do `:locals`:

```
<%= render partial: 'room', locals: {the_room: current_user.rooms.first} %>
```

Nesse caso, dentro da *partial*, teremos acesso a variável `the_room`, contendo `current_user.rooms.first`. O `locals` é útil para casos em que a *partial* depende de outras variáveis.

Agora vamos trabalhar no CSS. O que temos que fazer é extrair o *mixin* `shadow` do `default.css.scss` para um arquivo separado para que possamos usar tanto no `default.css.scss` quanto no novo CSS. Para isso, vamos criar um novo CSS para a função de sombra (`app/assets/stylesheets/shadow.css.scss`):

```
@mixin shadow($color, $x, $y, $radius) {
  -moz-box-shadow:    $color $x $y $radius;
  -webkit-box-shadow: $color $x $y $radius;
  box-shadow:         $color $x $y $radius;
}
```

Depois de apagar o *mixin* do `default.css.scss`, temos que incluir a função que acabamos de extrair. Fazemos isso com o uso do `@import`:

```
@import "shadow";

$header-height: 55px;
$content-width: 700px;

...
```

E para finalizar o CSS, vamos criar estilo para quartos (`app/assets/stylesheets/room.css.scss`):

```
@import "shadow";

.room {
  background-color: white;
  padding: 20px 25px;
```

```
  margin-top: 10px;
  @include shadow(#ccc, 0, 3px, 6px);
}

.room h2 {
  display: inline;
  a { font-size: 1.3em; }
}
```

Pronto. A próxima parte é o I18n (`config/locales/pt-BR.yml`):

```
pt-BR:
  # ...

  dialogs:
    destroy: 'Você tem certeza que quer remover?'

  rooms:
    index:
      title: 'Quartos disponíveis'
    room:
      owner_html: '&mdash; %{owner} (%{when})'
      edit: 'Editar'
      destroy: 'Remover'

  # ...
```

Lembre-se que o `_html` é necessário para que o I18n retorne entidades HTML (`mdash;`).

Por fim, a última alteração que precisamos fazer é atualizar o template da ação `show` (`app/views/users/show.html.erb`):

```
<%= render @room %>
```

Pronto, terminamos as regras de acesso! No próximo capítulo, vamos estudar como fazer avaliação de quartos, dando uma pontuação de 1 a 5 para um quarto, usando *AJAX* e associações um pouco mais complicadas do que vimos até então.

Capítulo 11

Avaliação de quartos, relacionamentos muitos para muitos e organização do código

Ruby foi construído para tornar os programadores mais felizes.
– Yukihiro "Matz" Matsumoto

A estrutura de dados e templates já está completa. O que vamos fazer nesse capítulo é possibilitar a avaliação de quartos, assim usuários do colcho.net podem observar as melhores localidades.

Primeiro, precisamos criar o modelo de avaliação. Em seguida, criaremos os relacionamentos entre os modelos `Usuario - User` e `Quarto - Room`, com validações. Com esses relacionamentos, vamos criar as ações de controle para alterar e criar avaliações.

No *front-end*, vamos alterar o template do quarto para incluir as opções de fazer a

avaliação, via AJAX. Para isso funcionar, vamos criar o controle com ações um pouco diferentes, para responder requisições AJAX. O resultado final será o seguinte:

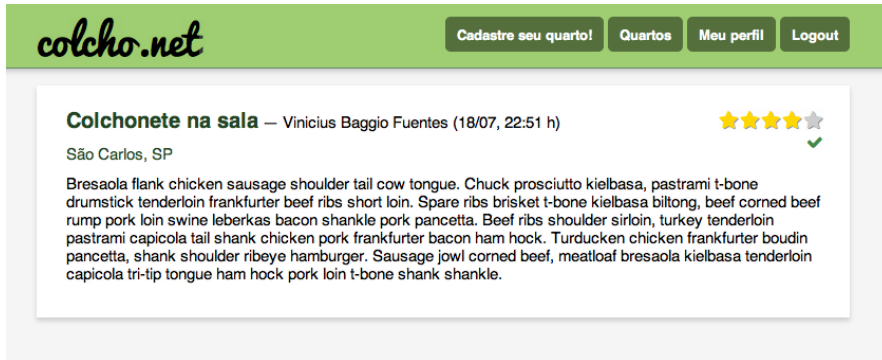



Figura 11.1: Estrelas para avaliação

Por fim, na listagem de todos os quartos, vamos aprender a usar as funções de cálculo (nesse caso, média de pontuação) do Rails e também vamos ver como executar consultas SQL mais complicadas.

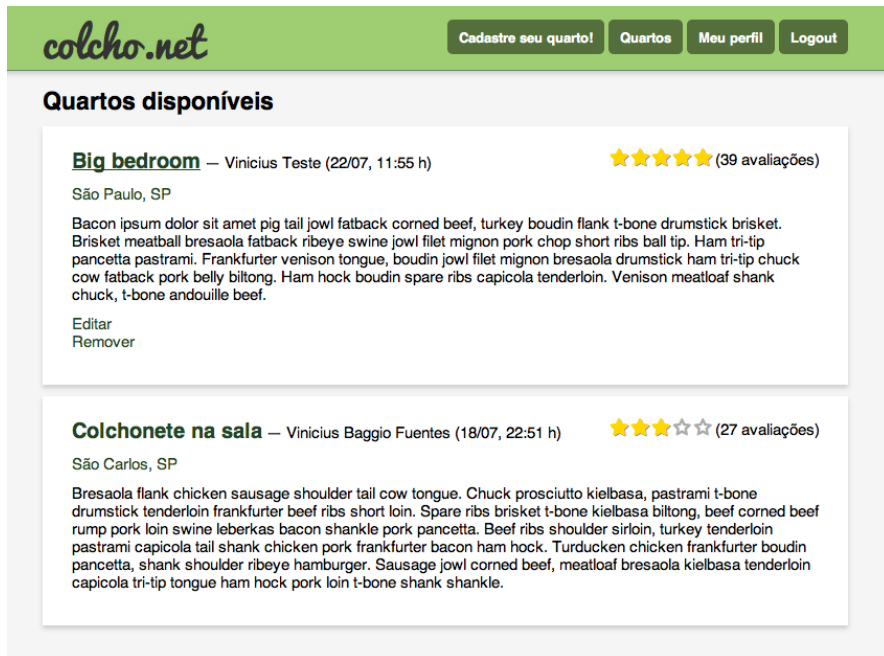



Figura 11.2: Avaliações na listagem com estrelas

11.1 Relacionamentos muitos-para-muitos

Uma avaliação é um modelo que pertence a um quarto e um usuário ao mesmo tempo. Isso significa que para uma avaliação ser única, ela depende de duas chaves estrangeiras: uma para o quarto e outra para o usuário avaliador.

Para ter esse resultado, vamos:

1) Criar modelo `Review` com chaves estrangeiras `user_id` e `room_id`, além de outros campos;

2) Criar índice para garantir unicidade do par `user_id` e `room_id`, ou seja, um usuário não pode avaliar um mesmo quarto mais de uma vez;

3) Criar validações no novo modelo, como por exemplo, não permitir que o usuário avalie o seu próprio quarto;

4) Criar o relacionamento no modelo quarto;

5) Criar o relacionamento no modelo usuário.

Criando chaves estrangeiras

Vamos usar o gerador para gerar o modelo `Review`, já com os relacionamentos:

```
$ rails g model review user:references room:references points:integer

      invoke  active_record
      create    db/migrate/20130719081519_create_reviews.rb
      create    app/models/review.rb
      invoke    test_unit
      create      test/unit/review_test.rb
      create      test/fixtures/reviews.yml
```

A migração gerada será assim:

```
class CreateReviews < ActiveRecord::Migration
  def change
    create_table :reviews do |t|
      t.belongs_to :user, index: true
      t.belongs_to :room, index: true
      t.integer :points

      t.timestamps
    end
  end
end
```

Precisamos alterar essa migração, pois vamos tirar os índices individuais em cada chave estrangeira e vamos colocar o índice no par `[user_id, room_id]`:

```
class CreateReviews < ActiveRecord::Migration
  def change
    create_table :reviews do |t|
      t.belongs_to :user
      t.belongs_to :room
      t.integer :points

      t.index [:user_id, :room_id], unique: true

      t.timestamps
```

```
    end
  end
end
```

Para efetivar essas alterações no banco de dados, executamos o comando `rake db:migrate`:

```
$ rake db:migrate
==  CreateReviews: migrating ====================
-- create_table(:reviews)
   -> 0.0094s
==  CreateReviews: migrated (0.0095s) ===========
```

Nenhuma novidade até então. O modelo `Review` (`app/models/review.rb`) também não tem muitos segredos -- o Rails até já criou o modelo com os `belongs_to` necessários:

```
class Review < ActiveRecord::Base
  belongs_to :user
  belongs_to :room
end
```

Vamos usar uma funcionalidade do Rails chamada *counter cache*, ou seja, *cache* de contadores. Como precisaremos calcular o número de avaliações, o Rails já guarda esse valor pré-calculado em uma coluna do banco de dados automaticamente, desde que usemos a opção `counter_cache: true` no `belongs_to`:

```
class Review < ActiveRecord::Base
  belongs_to :user
  belongs_to :room, counter_cache: true
end
```

Precisamos criar uma nova coluna na tabela de quartos de modo a guardar essa contagem:

```
$ rails g migration add_counter_cache_to_rooms reviews_count:integer
  invoke  active_record
  create    db/migrate/20130719081756_add_counter_cache_to_rooms.rb

$ rake db:migrate
==  AddCounterCacheToRooms: migrating =============
-- add_column(:rooms, :reviews_count, :integer)
```

```
   -> 0.0033s
==  AddCounterCacheToRooms: migrated (0.0034s) ====
```

Porém, isso não é suficiente. Precisamos colocar algumas validações no modelo de avaliações:

```
class Review < ActiveRecord::Base
  # Criamos um Array de 5 elementos, ao invés de range.
  POINTS = (1..5).to_a

  belongs_to :user
  belongs_to :room, counter_cache: true

  validates_uniqueness_of :user_id, scope: :room_id
  validates_presence_of :points, :user_id, :room_id
  validates_inclusion_of :points, in: POINTS
end
```

Essas validações são bem similares com as que já vimos, com exceção da opção `scope` aplicada à validação `uniqueness`. Ela limita o escopo em que a verificação de unicidade ocorre, ou seja, neste exemplo, o `user_id` pode repetir caso o `room_id` seja diferente.

Agora vamos ao modelo `Room` (`app/models/room.rb`). Nele vamos criar o relacionamento um-para-muitos: um quarto possui muitas avaliações.

```
class Room < ActiveRecord::Base
  has_many :reviews
  belongs_to :user

  #...
end
```

Relacionamentos de ponta-a-ponta

Em relacionamentos muitos-para-muitos, é comum precisarmos acessar o modelo da outra ponta da tabela de ligação, ou seja, no Colcho.net, é interessante um usuário saber todos os quartos que ele tem algum voto. A ideia é juntar todos os registros de avaliação que o usuário possui e, através desses registros, buscar os quartos.

O `ActiveRecord` te ajuda a resolver esse problema. Uma vez definido o relacionamento `has_many :reviews`, podemos criar o seguinte relacionamento:

```
class Room < ActiveRecord::Base
  # É necessário definir o has_many primeiro!
  has_many :reviews
  has_many :reviewed_rooms, through: :reviews, source: :room

  # ...
end
```

Esse exemplo cria o relacionamento `reviewed_rooms` que, através de todas as avaliações que um usuário tem (especificado pela opção `:through`), irá buscar o relacionamento `:room`, retornando todos os quartos que satisfazem estes relacionamentos.

Por fim, vamos criar o relacionamento no modelo `User` (`app/models/user.rb`), para que possamos ter acesso às avaliações daquele usuário:

```
class User < ActiveRecord::Base
  EMAIL_REGEXP = /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/

  has_many :reviews
  has_many :rooms


  # ...
end
```

Pronto! Todos os relacionamentos foram criados. Antes de continuar, veja o

exemplo a seguir:

```
# Lembre-se do sandbox para evitar a perda de dados!
# Para usar o sandbox, basta fazer "rails c --sandbox"

room = Room.last
# => #<Room id: 5, ... >

review = room.reviews.build points: 3
review.user = User.last
review.save
# => #<Review id: 1, ... >

Review.all
# => [#<Review id: 1, ... >]

room.destroy
# => #<Room id: 5, >

r = Review.first
# => #<Review id: 1, >

r.room
# => nil
```

O que acontece é que removemos o objeto `room`, porém ainda temos referências inválidas no banco de dados. Para arrumar isso, poderíamos criar um *callback* no momento que um objeto está sendo destruído (`after_delete`) e remover os objetos referenciados. Essa solução funciona, porém, o `ActiveRecord` facilita a nossa vida.

11.2 Removendo objetos sem deixar rastros

No momento que vamos destruir um objeto, é possível destruir também objetos relacionados. Para isso, adicionamos uma opção nas associações que queremos remover quando o objeto for destruído, chamada `dependent`. Seu comportamento é muito parecido com a opção `CASCADE` do SQL:

- `destroy` - Executa `#destroy` em todos os objetos associados, executando os *callbacks* de cada um;

- `delete_all` - Deleta os objetos associados via SQL apenas, sem execução de *callbacks*;
- `nullify` - Apenas marca as chaves estrangeiras dos objetos relacionados com `NULL`;
- `restrict` - Impede a remoção do objeto se houver objetos relacionados.

Nesse caso, queremos usar a opção `:destroy`, para que os *callbacks* sejam de fato chamados, precisamos destruir todas as avaliações associadas ao usuário quando ele é destruído. Para isso, basta colocar uma opção no `has_many`. Veja como deve ficar o modelo `User` (`app/models/user.rb`):

```
class User < ActiveRecord::Base
  EMAIL_REGEXP = /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/

  # Aproveite a oportunidade para atualizar o outro
  # relacionamento:
  has_many :rooms, dependent: :destroy
  has_many :reviews, dependent: :destroy

  # ...
end
```

E, por fim, no modelo `Room` (`app/models/room.rb`):

```
class Room < ActiveRecord::Base
  has_many :reviews, dependent: :destroy
  belongs_to :user

  # ...
end
```

Diferença entre `#destroy` e `#delete`

O `ActiveRecord` possui dois métodos para destruir objetos no banco de dados: o `#destroy` e o `#delete`. É muito importante lembrar que eles possuem comportamentos **diferentes**.

O `#destroy` é o método que normalmente deve ser usado. Ele executa todos os *callbacks* e deleta o objeto no banco de dados. O `#delete`, por sua vez, apenas executa o `DELETE` no banco de dados. Isso significa que nenhum *callback* e nenhum `dependent: destroy` serão executados.

Cuidado com o `dependent: :destroy`

Apesar da grande conveniência, o `dependent: :destroy` pode ser perigoso. Como o `ActiveRecord` tem que instanciar cada objeto e chamar o método `#destroy`, que por sua vez, pode ter seus relacionamentos, que vai instanciar todos os objetos, e... Deu pra entender onde isso vai parar, né?

Além de lento, esse procedimento pode ser perigoso. Se o seu banco de dados tiver muitos registros, o Rails irá instanciar um objeto `ActiveRecord` para cada linha, resultando em muita alocação de memória e muitas interrupções do *garbage collector* para limpar objetos não usados. Pior, seu sistema pode entrar em *swap* e parar de responder.

Em situações assim, você pode considerar uma das seguintes soluções:

- Marcar os objetos a serem removidos por um serviço que executa de tempos em tempos;
- Usar `dependent: delete` e usar chaves estrangeiras com `on delete cascade`, no próprio banco, fazendo com que o banco de dados fique responsável pela limpeza dos dados.

11.3 Criando avaliações com pitadas de AJAX

Pronto, agora que temos o modelo de avaliação, vamos implementar o *front-end*, usar uma nova fonte e incluir o JavaScript e chamadas AJAX.

Para isso, primeiro vamos criar o ponto de entrada para o recurso `Review`. Como um `Room` possui muitos `reviews`, vamos montar a URL de modo a representar esse relacionamento. A URL ficará da seguinte forma: `/rooms/:room_id/reviews`. Isso dará ao nosso projeto uma noção de que um quarto possui pelo menos uma avaliação.

O Rails representa essas rotas através do aninhamento de `resources`. Veja como ficará a rota para `reviews`:

```
Colchonet::Application.routes.draw do
  #...

  resources :rooms do
    resources :reviews, only: [:create, :update]
  end

  # ...
end
```

Vejamos o resultado executando `rake routes`:

```
room_reviews POST (/:locale)/rooms/:room_id/reviews(.:format)
                  reviews#create
                  {:locale=>/en|pt\-BR/}

room_review PUT   (/:locale)/rooms/:room_id/reviews/:id(.:format)
                  rooms/reviews#update
                  {:locale=>/en|pt\-BR/}
```

Seguindo a convenção, o Rails tentará carregar o controle como `ReviewsController` em `app/controllers/reviews_controller.rb`. Porém, depois de alguns meses ou anos de projetos, deixar todos os controles nessa pasta fica uma bagunça sem tamanho. Imagine uma pasta com mais de 100 controles, cada um em uma rota diferente.

Por essa razão, organizaremos controles que possuem aninhamento em módulos distintos. As principais vantagens dessa prática são: extrair comportamento comum,

como buscar o elemento a qual o recurso pertence (por exemplo, buscar o objeto `room` no controle de avaliações) ou filtros, e a organização dos arquivos.

Para modularizar componentes, o Rails requer que você crie uma subpasta com o nome do módulo e então ele será automaticamente definido para você. Aplicando a este caso, queremos criar o módulo `rooms`. Para isso, basta criar a pasta `app/controllers/rooms`. Em seguida, todo controle dentro dessa pasta deverá ser definido de acordo com o módulo. Para o caso do controle `reviews`, devemos criar o arquivo `app/controllers/rooms/reviews_controllers.rb`. Nesse arquivo, a classe a ser definida será `Rooms::ReviewsController`. As rotas, porém, necessitam de alterações para poder reconhecer o módulo.

Vamos alterar a rota que criamos de forma que o Rails busque o controle na pasta `app/controllers/rooms/` e a classe do controle seja `Rooms::ReviewsController`. Para isso, basta definir a opção `module`:

```
Colchonet::Application.routes.draw do
  #...

  resources :rooms do
    resources :reviews, only: [:create, :update], module: :rooms
  end

  # ...
end
```

O que precisamos fazer no controle de `reviews` é criar uma nova entrada no banco de dados quando um usuário fizer uma nova avaliação. Precisamos também garantir que, quando o usuário faça uma nova avaliação, apenas atualizaremos a entrada já existente. É possível fazê-lo com o que já vimos até agora, mas é possível realizar essa operação de forma que o código fique bem mais limpo.

A solução mais natural é fazer uma busca e, se o objeto não existir, criar um novo. Em seguida, atualizamos o objeto para ter o novo valor. É possível fazer isso de uma só vez, usando o método `find_or_initialize_by`. Esse método fará exatamente o que precisamos: buscará o objeto e, se não existir, criará um novo objeto com o valor preenchido. Com este objeto, podemos preencher os outros atributos e salvá-lo no banco de dados. Note que esse método não salva o modelo no banco de dados. Para tal, você pode usar o `find_or_create_by`.

A ação `create` ficará da seguinte forma: faremos a busca em todas as avaliações do quarto especificado a partir do usuário logado (`current_user`). Em seguida,

preencheremos o restante dos atributos vindos do formulário.
 (app/controllers/rooms/reviews_controller.rb).

```
class Rooms::ReviewsController < ApplicationController
  before_filter :require_authentication

  def create
    review = room.reviews.
      find_or_initialize_by(user_id: current_user.id)

    review.update!(review_params)

    head :ok
  end

  def update
    create
  end

  private

  def room
    @room ||= Room.find(params[:room_id])
  end

  def review_params
    params.
      require(:review).permit(:points)
  end
end
```

Métodos obsoletos no Rails 4

A maioria dos `dynamic finders` se tornaram obsoletos no Rails 4. Por exemplo, não será mais possível chamar o método `find_by_email` no modelo `User`, como vimos anteriormente. O mesmo vale para os métodos `%find_or_create_by...` e `find_or_initialize_by...`.

Como vamos responder apenas a requisições AJAX, não precisamos renderizar nenhum conteúdo. Portanto, apenas respondemos com o código HTTP `201 Created` quando há sucesso, por isso usamos o método `head`. Esse método retorna o cabeçalho HTTP contendo o código de resposta (no caso, 201) e como corpo, apenas um espaço em branco.

Curiosidade: método head e o famigerado espaço em branco

O método `head` não precisaria emitir um espaço em branco. Porém, versões antigas do Internet Explorer ignoram a resposta do servidor caso o corpo esteja vazio.

Dependendo da biblioteca JavaScript que você usa, isso pode trazer um problema. Bibliotecas que fazem AJAX podem tentar fazer a interpretação do resultado como se fosse objetos JSON, resultando em falhas no script e interrompendo a execução.

Nesse caso, como o único *input* do usuário é um valor pré-selecionado, não devemos encontrar erros. Se por um acaso encontrarmos, há algum problema em nosso projeto e portanto usamos o método *bang* no `#update!`, de forma a disparar exceções e, portanto, facilitar a depuração.

O comportamento do `update` deverá ser o mesmo do `create`, portanto apenas chamamos o método. A única diferença é que objeto será encontrado no `find_or_initialize_by`.

Falha silenciosa vs. falha "barulhenta"

Quando desenvolvemos aplicações, temos a tendência de tentar tratar ou silenciar erros, evitando que usuários sejam apresentados a uma página de erro.

Porém, é importante salientar que, quando fazemos isso, fica mais difícil descobrir que algo está errado e tentar caçar algum *bug* misterioso. Quando algo inesperado acontece, é melhor ver um *stack trace* do que tentar adivinhar, fazendo *debug* por horas para descobrir porque uma variável está `nil`, por exemplo.

Dica: Reuso de código em controles

Não precisamos criar abstrações no nosso exemplo. Porém, na necessidade de compartilhar código entre vários controles em um mesmo módulo, podemos criar uma classe chamada `Rooms::BaseController` e nela colocar filtros e outros métodos auxiliares:

```
class Rooms::BaseController < ApplicationController
  before_filter :require_authentication
  private

  def room
    @room ||= Room.find(params[:id])
  end
end
```

Em seguida, basta herdar desse controle e o comportamento será compartilhado:

```
class Rooms::ReviewsController < Rooms::BaseController
  def create
    review = room.reviews.
      find_or_initialize_by(user_id: current_user.id)

    # ...
  end
end
```

Por fim, vamos acertar o controle `RoomsController` (`app/controllers/rooms_controller.rb`) para construir um objeto de avaliação a ser usado no template. Faremos isso para que os *helpers* de formulário do Rails já popule os campo `user_id` apropriadamente.

```
class RoomsController < ApplicationController
  # ...

  def show
    @room = Room.find(params[:id])
```

```
    if user_signed_in?
      @user_review = @room.reviews.
        find_or_initialize_by(user_id: current_user.id)
    end
  end

  #...
end
```

Controles prontos, vamos aos *templates*. Vamos colocar as tradicionais estrelas de avaliação, mas a priori focaremos na funcionalidade. Vamos usar *radio buttons* para que o usuário escolha a pontuação, de 1 a 5:

Figura 11.3: Avaliação de quartos com radio buttons

Para fazer isso, vamos criar um formulário na *partial* de quartos (app/views/rooms/_room.html.erb):

```
<article class="room">
  <h2><%= link_to room.title, room %></h2>

  <%= render partial: 'review', object: @user_review %>

  <span class="created">
    <%= t '.owner_html',
      owner: room.user.full_name,
      when: l(room.created_at, format: :short) %>
  </span>
```

```
  ...
</article>
```

Criemos a *partial* `review` (`app/views/rooms/_review.html.erb`). Nela, precisamos criar um formulário contendo os *radio buttons*, caso o usuário esteja logado. Caso contrário, exibimos uma mensagem dizendo que essa operação só está disponível para usuários já autenticados:

```
<section class="review">
  <% if user_signed_in? %>
    <%= form_for [review.room, review] do |f| %>
      <% Review::POINTS.each do |point| %>
        <%= f.radio_button :points, point  %>
        <%= f.label :points, point, value: point %>
      <% end %>

      <%= f.submit %>
    <% end %>
  <% else %>
    <span class="login_required">
      <%= t('.login_to_review') %>
    </span>
  <% end %>
</section>
```

A primeira notável diferença é a construção da rota para o `form_for`. Como o recurso "avaliação" é aninhado ao recurso "quarto", é necessário identificar a qual quarto pertence a nova avaliação que criamos no controle de quartos (lembre-se de como as rotas foram criadas). Usando a notação de Array, dizemos ao Rails todas as dependências da rota.

Note que isso causa um problema com a listagem de quartos (ação `index`). A razão é que a listagem usa a mesma *partial* de quartos, porém não possui a revisão pré construída. Vamos deixar esse problema de lado por enquanto até concluirmos a exibição de quartos.

Rotas com *namespaces*

É possível criar rotas com *namespaces*, ou seja, um nome que na verdade não representa um recurso, mas que divide a aplicação em "módulos". Por exemplo, uma área de administração (ou "admin") pode ser um *namespace*. Para declará-los nas rotas, basta fazer:

```
namespace :admin do
  resources :products
end
```

Para você identificar *namespaces* nos formulários, você deve também usar a notação de Array:

```
<%= form_for [:admin, @product] do |f| %>
 ...
```

Quando criamos formulários em HTML, os campos de *label* possuem um atributo chamado `for`, que deve possuir o `name` ou `id` do elemento à que este *label* se associa. Fazendo isso, clicar no texto do *label* irá selecionar o campo de texto, se associado com um campo de texto, selecionar um elemento de um grupo de *radio buttons* e assim por diante.

Para tornar *radio buttons* em um mesmo grupo (ou seja, selecionando um irá desmarcar o outro), é necessário usar um mesmo `name`. Isso quebra com a forma de que o `label` funciona, ou seja, todos os *labels* de um mesmo grupo, se usando os *helpers* do Rails, iriam apontar para um mesmo botão, deixando de funcionar da maneira esperada.

É aí que entra a opção `:value` do *helper* `label`. Você deve usar essa opção tanto para *labels* em elementos do tipo *radio button* ou *check boxes*. O *helper* irá construir o atributo `for` apontando para o elemento correto.

Por fim, adicione as seguintes regras CSS para os botões alinharem da forma correta e estilizar o texto de login (`app/assets/stylesheets/room.css.scss`):

```
// Usamos o #content para aumentar a especificidade do seletor,
// ou seja, tornar essa regra mais importante que outras.
#content .review form {
  margin: 0;
```

```
}

#content .review {
  margin: 0;
  padding: 0;
  float: right;
  border: none;
}

.review label {
  display: inline;
}

.review .login_required {
  font-size: 0.8em;
  color: #666;
  font-variant: small-caps;
}
```

Figura 11.4: Texto de login necessário

Traduzimos o modelo `Review`:

```
pt-BR:
  # ...
  activerecord:
    models:
      room: Quarto
      user: Usuário
      review: Avaliação
```

E traduzimos a mensagem de login:

```
pt-BR:
  ...

  rooms:
    index:
      title: 'Quartos disponíveis'
    room:
      owner_html: '&mdash; %{owner} (%{when})'
      edit: 'Editar'
      destroy: 'Remover'
    review:
      login_to_review: Faça o login para avaliar quartos
```

11.4 Diga adeus a regras complexas de apresentação: use presenters

Tem algo que muito me incomoda nesse código, para ser sincero. Temos uma regra de *template* repetida tanto no `RoomsController` quanto no *template*.

No `RoomsController`:

```
if user_signed_in?
  @user_review = @room.reviews.
    find_or_initialize_by(user_id: current_user.id)
end
```

E no *template*:

```
<% if user_signed_in? %>
  <%= form_for [review.room, review] do |f| %>
  <%# ... %>
<% else %>
  <%# ... %>
<% end %>
```

Para resolver esse problema, vamos criar uma classe que vamos usar tanto nos *templates* quanto no controle. Os *presenters*, ou apresentadores, são responsáveis por fazer essa ligação, resolvendo o problema de *templates* ou controles complexos.

Vamos criar então o `RoomPresenter`. Crie o arquivo e a pasta `app/presenters/room_presenter.rb`.

A ideia dessa classe é a seguinte: podemos passar um quarto a qual a avaliação irá pertencer e o contexto que o *presenter* se aplica: pode ser tanto um template quanto o próprio controle. Precisamos deste contexto para saber se o usuário está logado, por exemplo. Outras informações são usadas pelo Rails para renderizar links de modelos (`to_param` e `model_name`), e o restante servirá para simplificar o template.

```
class RoomPresenter
  delegate :user, :created_at, :description, :location, :title,
    :to_param, to: :@room

  def self.model_name
    Room.model_name
  end

  def initialize(room, context, show_form=true)
    @context = context
    @room = room
    @show_form = show_form
  end

  def can_review?
    @context.user_signed_in?
  end

  def show_form?
    @show_form
  end

  def review
    @review ||= @room.reviews.
      find_or_initialize_by(user_id: @context.current_user.id)
  end

  def review_route
    [@room, review]
  end

  def route
    @room
  end
```

```
  def review_points
    Review::POINTS
  end

  # Faz com que a partial 'room' seja renderizada quando chamamos o 'render'
  # com o objeto da classe room presenter.
  def to_partial_path
    'room'
  end
end
```

As ações `show` e `index` do controle `RoomsController` (`app/controllers/rooms_controller`) ficam da seguinte forma:

```
class RoomsController < ApplicationController
  # ...

  def index
    # O método #map, de coleções, retornará um novo Array
    # contendo o resultado do bloco. Dessa forma, para cada
    # quarto, retornaremos o presenter equivalente.
    @rooms = Room.most_recent.map do |room|
      # Não exibiremos o formulário na listagem
      RoomPresenter.new(room, self, false)
    end
  end

  def show
    room_model = Room.find(params[:id])
    @room = RoomPresenter.new(room_model, self)
  end
end
```

A *partial* `room` (`app/views/rooms/_room.html.erb`), por sua vez, fica:

```
<article class="room">
  <h2><%= link_to room.title, room.route %></h2>

  <%= render partial: 'review', locals: {room: room} %>

  <span class="created">
```

```
  <%# ... $>
  <% if belongs_to_user(room) %>
    <ul>
      <li><%= link_to t('.edit'), edit_room_path(room.route) %></li>
      <li><%= link_to t('.destroy'), room_path(room.route),
        method: :delete,
        data: {confirm: t('dialogs.destroy')} %>
      </li>
    </ul>
  <% end %>
</article>
```

Por fim, vamos extrair o formulário de `review` (`app/views/rooms/_review.html.erb`), em uma nova *partial*, `review_form` (`app/views/rooms/_review_form.html.erb`).

A partial `app/views/rooms/_review.html.erb` ficará assim:

```
<section class="review">
  <% if room.show_form? %>
    <%= render partial: 'review_form', locals: {room: room} %>
  <% end %>
</section>
```

E a partial `app/views/rooms/_review_form.html.erb` ficará como a seguir:

```
<% if room.can_review? %>
  <%= form_for room.review_route do |f| %>
    <% room.review_points.each do |point| %>
      <%= f.radio_button :points, point  %>
      <%= f.label :points, point, value: point %>
    <% end %>

    <%= f.submit %>
  <% end %>
<% else %>
  <span class="login_required">
    <%= t('.login_to_review') %>
  </span>
<% end %>
```

Esse trabalho com as partials facilitará o nosso trabalho toda vez que quisermos reusar a revisão de quartos.

11.5 jQuery e Rails: fazer requisições AJAX ficou muito fácil

O código que temos atualmente cria avaliações, porém clicar no botão 'Criar Avaliação' irá nos exibir uma página em branco. A ação foi projetada para que façamos a requisição via AJAX, e é isso que vamos fazer agora.

Você deve estar pensando: "bom, vamos primeiro instalar nosso framework Javascript preferido (*aham* jQuery) e usá-lo para facilitar nossa vida". Certíssimo! Atualmente é difícil o site que use JavaScript e não possua o jQuery, para o bem ou para o mal.

O Rails, como um framework que quer te ajudar a colocar o seu site no ar, já instala o jQuery (http://jquery.com/) em sua aplicação. E mais ainda, o Rails ainda integra o jQuery em muito dos *helpers*, incluindo formulários via AJAX.

Portanto, para fazer nosso formulário usar AJAX e enviar as requisições assincronamente, basta alterar uma linha.

Altere o `form_for` na *partial* do formulário de *review* (`app/views/rooms/_review_form.html.erb`) para incluir a opção `remote: true`:

```
<% if room.can_review? %>
  <%= form_for room.review_route, remote: true do |f| %>
    <%# ... %>
  <% end %>
<% else %>
  <%# ... %>
<% end %>
```

Pronto! Ao usar o `remote: true`, o Rails criará um atributo `data` chamado `data-remote`. A partir daí, um script chamado "Unobtrusive scripting adapter", ou adaptador para script não-intrusivo, irá observar o evento `submit` do formulário, capturá-lo e enviar todos os dados via AJAX, ao invés da forma tradicional.

Esse adaptador está intimamente ligado ao jQuery, mas existem outras implementações dele, fazendo essa ligação a outros frameworks, como o prototype.js (http://prototypejs.org/) .

Se você ainda desejar usar outro framework de sua preferência, não é difícil criar o seu próprio, basta remover a entrada `gem 'jquery-rails'` do `Gemfile`, que é quem habilita o jQuery no projeto.

O adaptador também disponibiliza alguns eventos para que possamos registrar *callbacks* e fazer alguma customização de comportamento. Vamos usar esses eventos para travar o botão do formulário durante o envio da requisição (`ajax:beforeSend`) e sinalizar sucesso (`ajax:success`) ou erro (`ajax:error`) ao usuário. A lista completa de eventos é:

- `ajax:before` - Antes de preparar a requisição AJAX (ou seja, antes de capturar os campos do formulário e preparar as opções de envio da requisição);
- `ajax:beforeSend` - Chamado antes de enviar a requisição, porém com tudo já pronto para envio;
- `ajax:success` - Depois do término da requisição e a resposta foi bem sucedida;
- `ajax:error` - Depois do término da requisição e a resposta foi mal sucedida;
- `ajax:complete` - Chamado depois do término da requisição, não importando o resultado.

Desenvolvendo JavaScript manualmente

Se você preferir, é possível não usar o adaptador do Rails e fazer tudo "na mão". O que temos, por fim, é a própria biblioteca jQuery, então podemos usá-la da forma que quisermos.

Por fim, se você quiser, pode usar CoffeeScript, bastando adicionar a extensão `.coffee` e escrever o código equivalente. Usaremos JavaScript no livro.

Crie o arquivo `app/assets/javascripts/room.js`. Nele, criaremos *callbacks* para três eventos: `ajax:beforeSend` (e **não** `ajax:before`), `ajax:error` e `ajax:success`. No evento `ajax:beforeSend`, desativamos todos os *inputs*, pois eles não serão mais operacionais (não podemos votar mais de uma vez). Se tudo der certo, substituímos o botão por um ícone de sucesso ("v") e se der errado, por um "x":

```
$(function() {
  var $review = $('.review');
```

```
  $review.on('ajax:beforeSend', function() {
    $(this).find('input').attr('disabled', true);
  });

  $review.on('ajax:error', function() {
    replaceButton(this, 'icon-remove', '#B94A48');
  });

  $review.on('ajax:success', function() {
    replaceButton(this, 'icon-ok', '#468847');
  });

  function replaceButton(container, icon_class, color) {
    $(container).find('input:submit').
        replaceWith($('<i/>').
                      addClass(icon_class).
                      css('color', color));
  };
});
```

11.6 Média de avaliações usando agregações

Estamos quase terminando a funcionalidade de avaliações. A última coisa que vamos fazer antes das mudanças visuais é exibir a nota média de avaliações do quarto.

Para isso, criaremos um método no modelo de avaliações para retornar o valor da média de pontos, usando uma função de cálculo do `ActiveRecord`. As funções de cálculo são:

- `average` - média;
- `minimum` - mínimo;
- `maximum` - máximo;
- `count` - contagem;
- `sum` - soma.

Nesses métodos, você deverá passar o nome do campo cujos valores serão calculados. Poderá passar também algumas opções de configuração. Veja alguns exemplos:

```
Review.average('points')
#  SELECT AVG("reviews"."points") AS avg_id FROM "reviews"
# => #<BigDecimal:7f8b34bd8ed8,'0.21E1',18(45)>

Review.average('points').to_f
# => 2.1

# Funciona com escopos!
Room.first.reviews.average('points').to_f
# SELECT AVG("reviews"."points") AS avg_id
#   FROM "reviews" WHERE "reviews"."room_id" = 4
#
# => 2.25

Review.average(:points, group: :room)
#  SELECT AVG("reviews"."points") AS average_points,
#        room_id AS room_id FROM "reviews"  GROUP BY room_id
#
# => {#<Room id: 4, ...>=>#<BigDecimal:7f8b349ab318,'0.225E1',18(45)>,
#     #<Room id: 5, ...>=>#<BigDecimal:7f8b349aa7b0,'0.2E1',9(45)>}

Review.maximum(:points)
# SELECT MAX("reviews"."points") AS max_id FROM "reviews"
# => 5

Review.minimum(:points)
# SELECT MIN("reviews"."points") AS min_id FROM "reviews"
# => 1

Review.count
# SELECT COUNT(*) FROM "reviews"
# => 10
```

A principal vantagem desses métodos de cálculo é que, como você pôde observar, eles são todos feitos via SQL e portanto serão beneficiados pelos índices do banco de dados e não há o custo de alocar um objeto na memória para cada registro.

Usando o método `.average` do `ActiveRecord`, calcularemos a média de estrelas a partir de uma coleção de avaliações. Usando o método `#round` de números, arrendondamos as estrelas para um número inteiro. Juntando esses dois métodos, vamos criar um método de classe no modelo `Review` (`app/models/review.rb`)

-- também aplicáveis a escopos.

```
class Review < ActiveRecord::Base
  # ...

  def self.stars
    (average(:points) || 0).round
  end
end
```

Veja exemplos de uso:

```
Room.first.reviews.stars
# => 2

Review.stars
# => 2
```

Vamos então usar esse método no *presenter* para quartos (`app/presenters/room_presenter.rb`), criando os métodos `#stars` e `total_reviews`:

```
class RoomPresenter
  # ...

  def review_points
    Review::POINTS
  end

  def stars
    @room.reviews.stars
  end

  def total_reviews
    @room.reviews.size
  end

  # ...
end
```

Dica: métodos de contagem

Como escopos se comportam como Arrays, eles podem causar problemas sem mesmo perceber. Arrays possuem três métodos que contam o número de objetos: `#length`, `#count` e `#size`. Todos eles funcionam da mesma maneira.

Porém, isso não se reflete em escopos e modelos `ActiveRecord`. Os três métodos existem, mas com comportamentos diferentes:

- `#length` - É o mesmo do Array, portanto faz com que o `ActiveRecord` busque todos os objetos no banco, instancie-os e depois faz a contagem;
- `#count` - Conta quantos objetos existem no banco de dados, fazendo uma consulta SQL (vimos agora pouco: faz parte dos métodos de cálculo) ou usando o *counter cache*;
- `#size` - Chama a contagem pelo método `#length` caso os objetos tenham sido carregados, caso contrário, conta via o método `#count`.

Dessa forma, sempre que for fazer contagem de objetos, prefira usar o `#size`. Se você precisar usar a coleção completa de objetos (cuidado com o uso de memória!), use o `#length`.

Vamos mostrar então no *template* de avaliações (`app/views/rooms/_review.html.erb`) esses números:

```
<section class="review">
  <% if room.show_form? %>
    <%= render partial: 'review_form', locals: {room: room} %>
  <% else %>
    <%= t '.stats', average: room.stars,
                    max: room.review_points.max,
                    count: room.total_reviews %>
  <% end %>
</section>
```

E no arquivo de I18n (`config/locales/pt-BR.yml`), usamos a funcionalidade para plurais:

```
pt-BR:
  # ...
  rooms:
    #...

    review_form:
      login_to_review: 'Faça o login para avaliar quartos'
    review:
      stats:
        zero: 'Não há avaliações'
        one: '%{average}/%{max} (1 avaliação)'
        other: '%{average}/%{max} (%{count} avaliações)'
```

O que acontece é que, quando usamos a chave `:count` na tradução, podemos criar mensagens diferenciadas para cada valor de `:count`: *zero* para zero, *one* para um e *other* para o restante, assim não precisamos nos preocupar em fazer regras para cada valor.

Figura 11.5: Avaliações com estatísticas básicas

11.7 Aplicações modernas usam fontes modernas

Por muito tempo, designers sofriam na criação de páginas Web. Limitados a não mais que 10 fontes, eles tinham que fazer milagres para tornar um site elegante e não cair em *clichés*. Porém, isso mudou completamente desde que os *browsers* passaram a aceitar fontes embutidas via a diretiva `@font-face`. Com ela, é possível que um site use uma fonte não instalada no sistema do usuário.

Além disso, o Google criou o serviço *Google Webfonts* (www.google.com/webfonts/), serviço gratuito e que possui diversas fontes de alta qualidade e é muito fácil de usar. É exatamente isso que estamos usando no Colcho.net para usar a fonte "Pacifico".

Recentemente, as *webfonts* tem sido usadas para outro propósito. Com a criação de dispositivos com altíssima densidade de *pixels* (o "Novo iPad" e os MacBooks com "Retina Display"), os browsers acabam escalonando as imagens para que elas fiquem no tamanho configurado. Isso resulta em defeitos e degradação da qualidade das imagens em um site. Veja o exemplo a seguir:

Figura 11.6: TV5.org: Site não preparado para alta densidade

Veja a diferença entre a qualidade da fonte, que é um elemento facilmente escalonável e as imagens. Quando elas são ampliadas, muitos defeitos visuais ocorrem. Compare com o exemplo do site CSS Tricks (http://www.css-tricks.com) , que é otimizado para "Retina Display":

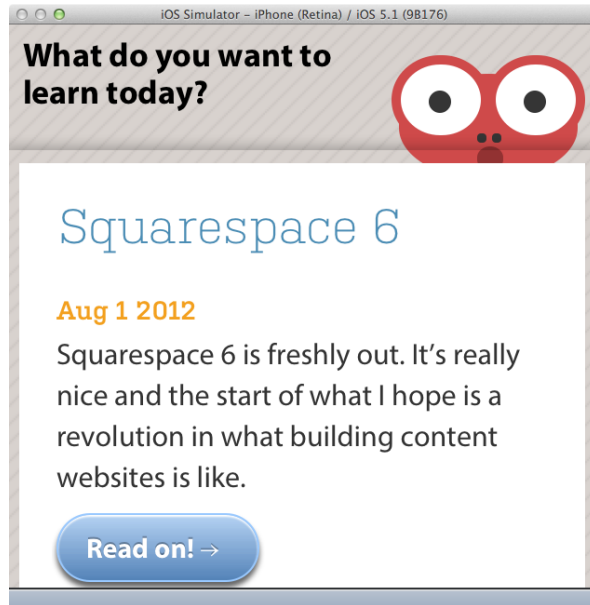



Figura 11.7: CSS-Tricks.com: Otimizado para alta densidade

Alcançar esse resultado não é simples, é necessário aplicar vários truques com CSS e repetir imagens de forma a ter uma versão com pouca e outra com alta densidade de pixels, ou através do uso de imagens SVG (em vetor). Outra solução é transformar seus ícones e imagens bastante utilizadas em fontes. Como as fontes são facilmente escalonáveis (pois são vetores, e não mapa de pixels) e relativamente simples de serem criadas, muitos designers e desenvolvedores de *front-end* estão optando por essa solução.

Existe uma fonte *open source* e disponível para uso comercial chamada "Font Awesome" (http://fortawesome.github.com/Font-Awesome/) . Além de possuir grande qualidade, ela possui vários pictogramas úteis e é de graça.

Portanto, antes de continuar com o desenvolvimento do site, vamos instalá-la no Colcho.net. Para isso, existe uma gem para que possamos usar a FontAwesome no Assets Pipeline. Adicione ao Gemfile:

```
gem "font-awesome-rails"
```

```
$ bundle
...
```

```
Installing font-awesome-rails (3.2.1.2)
...
```

Por fim, basta incluir a linha do FontAwesome no `application.css`, resultando em:

```
/*
 * ...
 *= require_self
 *= require_tree .
 *= require font-awesome
 */
```

Pronto, temos a Font Awesome instalada e pronta para uso.

11.8 Eu vejo estrelas - usando CSS e JavaScript para melhorar as avaliações

Com *back-end* e as interações prontos, vamos colocar a "Font Awesome" para uso: usaremos estrelas para mostrar as pontuações de 1 a 5 na listagem. Para isso, vamos mudar um pouco o *template*, o CSS e o I18n.

Comecemos pelo template de avaliações (`app/views/room/_review.html.erb`). Primeiro, criamos o número de estrelas preenchidas (`filled_star`) e depois criamos o número de estrelas não preenchidas (subtraindo o número de estrelas do quarto do total de 5). O resultado é o seguinte:

```
<section class="review">
  <% if room.show_form? %>
    <%= render partial: 'review_form', locals: {room: room} %>
  <% else %>
    <% room.stars.times do %>
      <span class="star filled_star"></span>
    <% end %>

    <% (room.review_points.max - room.stars).times do %>
      <span class="star empty_star"></span>
    <% end %>

    <%= t '.stats', count: room.total_reviews %>
```

```
  <% end %>
</section>
```

Por exemplo, no caso de 27 avaliações que resultam em 3 estrelas, o HTML resultante será:

```
<section class="review">
  <span class="star filled_star"></span>
  <span class="star filled_star"></span>
  <span class="star filled_star"></span>

  <span class="star empty_star"></span>
  <span class="star empty_star"></span>

  (27 avaliações)
</section>
```

Atualizamos também o I18n (`config/locales/pt-BR.yml`) para alterar a mensagem:

```
pt-BR:
  rooms:
    review_form:
      login_to_review: 'Faça o login para avaliar quartos'
    review:
      stats:
        one: '(1 avaliação)'
        other: '(%{count} avaliações)'
```

E, por fim, o CSS (`app/assets/stylesheets/room.css.scss`):

```
.review .star {
  font-family: "FontAwesome";
  font-size: 18px;
}

.review .filled_star {
  color: gold;
  text-shadow: 1px 1px #999;
  &:before { content: "\f005"; }
}
```

```
.review .empty_star {
  color: #aaa;
  &:before { content: "\f006"; }
}
```

Usamos o `:before` para adicionar um caractere especial da "Font Awesome" antes das estrelas. Depois dessas alterações, temos a listagem com estrelas:

Figura 11.8: Avaliações na listagem com estrelas

Agora, para finalizar, tornaremos a votação mais interessante, vamos usar as estrelas de modo que o usuário possa votar apenas no clique. Para isso, alteraremos o CSS de forma que o cursor do mouse indique ao usuário que ele pode interagir com a estrela, com diferentes cores e apresentações.

Para isso, primeiro altere o CSS do formulário (`app/assets/stylesheets/room.css.scss`):

```
#content .review {
  margin: 0;
  padding: 0;
```

```
  float: right;
  border: none;
  /* Adicione a seguinte linha: */
  position: relative;
}
```

Em seguida, adicione as regras para posicionar os símbolos de OK e erro em um lugar de forma que as estrelas não saiam do lugar quando o formulário for enviado.

```
.review .icon-ok,
.review .icon-remove {
  position: absolute;
  right: 0;
}
```

As próximas 2 regras são para estilizar a estrela de acordo com o novo HTML do formulário, que veremos em seguida.

```
.review label i {
  font-size: 18px;
  text-shadow: 1px 1px #999;
  cursor:pointer;
  color: #ccc;
}

.review label.toggled i {
  color: gold;
}
```

A última regra é para sumir com os botões *radio* e o "enviar".

```
.review form > input {
  display:none;
}
```

O novo template do formulário (`app/views/rooms/_review_form.html.erb`) deve ficar assim:

```
<% if room.can_review? %>
  <%= form_for room.review_route, remote: true do |f| %>
    <% room.review_points.each do |point| %>
      <%= f.radio_button :points, point  %>
```

```
      <%= f.label :points, value: point do %>
        <i class="icon-star"></i>
      <% end %>
    <% end %>

    <%= f.submit %>
  <% end %>
<% else %>
  <span class="login_required">
    <%= t('.login_to_review') %>
  </span>
<% end %>
```

A diferença está na maneira que construímos o *label*. Ao invés de colocarmos o valor, vamos simplesmente desenhar uma estrela, de acordo com o “Font-Awesome”.

Para encerrar essa funcionalidade, adicionamos o código JavaScript para alterar a cor das estrelas quando o usuário passar o mouse sobre elas, e com isso mostrar ao usuário que ele está de fato alterando a sua avaliação.

Para isso, adicione o seguinte código JavaScript no arquivo `app/assets/javascripts/rooms.js`:

```
$(function() {
  // ...

  function highlightStars(elem) {
    elem.parent().children('label').removeClass('toggled');
    elem.addClass('toggled').prevAll('label').addClass('toggled');
  }

  highlightStars($('.review input:checked + label'));

  var $stars = $('.review input:enabled ~ label');

  $stars.on('mouseenter', function() {
    highlightStars($(this));
  });

  $stars.on('mouseleave', function() {
    highlightStars($('.review input:checked + label'));
  });
```

```
  $('.review input').on('change', function() {
    $stars.off('mouseenter').off('mouseleave').off('click');
    $(this).parent('form').submit();
  });
});
```

A primeira função, `highlightStars`, é a responsável por adicionar e remover o destaque das estrelas. Baseado no elemento passado como parâmetro, primeiro remove-se o destaque de todas as estrelas (classe CSS `toggled`) e em seguida adiciona-se a mesma classe apenas ao elemento em destaque e os anteriores.

Usando essa função, ativamos as estrelas previamente selecionadas pelo usuário. Isso é importante para mostrar ao usuário que a ação dele teve efeito. Este seletor executa no momento que a página é carregada.

A primeira parte, `input:checked` irá retornar todos os *inputs* que estão marcados (válido somente para *check boxes* e *radio buttons*). Usando o `+`, retornamos apenas o primeiro objeto **imediatamente** ao redor deste *input*. Isso significa que o seletor irá retornar o *label* imediatamente ao lado de um *input* que está selecionado. Veja a seguir o resultado:

Figura 11.9: Formulário de avaliação enviado

```
var $stars = $('.review input:enabled ~ label');
```

Na linha anterior, selecionamos todos os *labels* (as estrelas) em *inputs* que estão habilitados. O comportamento é parecido com o seletor `input:checked + label`, porém dessa vez procuramos **todos** que estão habilitados. O objetivo é que só iremos dar destaque quando o formulário estiver habilitado, ou seja, apenas antes de ser enviado.

```
$stars.on('mouseenter', function() {
  highlightStars($(this));
});

$stars.on('mouseleave', function() {
  highlightStars($('.review input:checked + label'));
});
```

Esses blocos são responsáveis pelos eventos do mouse. O primeiro evento, o `mouseenter`, ocorre quando o usuário posiciona o mouse em cima de uma estrela. Nesse momento, vamos destacar a estrela clicada e as anteriores.

No evento `mouseleave`, vamos voltar ao estado inicial do formulário, ou seja, se já havia uma estrela marcada, tornamos ela marcada novamente.

Finalmente, temos o seguinte bloco:

```
$('.review input').on('change', function() {
  $stars.off('mouseenter').off('mouseleave').off('click');
  $(this).parent('form').submit();
});
```

O evento `change` é executado após o *click* em uma estrela. Nesse momento, desligamos todos os *Event handlers* que criamos, ou seja, desabilitamos a animação de estrelas e ativamos o evento de `submit` do formulário, fazendo o envio do formulário via AJAX.

11.9 Encerrando

Parabéns! Você perseverou até o fim das funcionalidades principais do Colcho.net! Foi um caminho longo e difícil, especialmente nesse capítulo. Mas você aprendeu a fazer **muita** coisa:

- Associações muitos-para-muitos;
- Remover objetos com segurança;
- Usar fontes e webfonts para deixar seu design bonito e escalonável;
- Customizar o Assets Pipeline;
- Novas funcionalidades do `ActiveRecord`;

- Organização de controles complexos;
- Organização de rotas complexas;
- Usar o `ActiveRecord` para fazer contas de maneira eficiente;
- Opções avançadas de I18n;
- Extração e organização de templates complexos;
- Uso de *presenters* para simplificar templates complexos;
- Usar jQuery e AJAX;

Depois de tudo isso, você já está preparado para criar sua própria aplicação do zero. Os conhecimentos vistos nesse capítulo já cobrem muitas funcionalidades usadas em aplicações de verdade, com bastante complexidade. É lógico que livro nenhum irá substituir a experiência de construir as suas próprias aplicações, mas agora você já tem a base para colocar suas ideias em prática.

No próximo capítulo, vamos colocar algumas funcionalidades satélites, úteis mas não essenciais para o funcionamento da aplicação. Essas funcionalidades, porém, são muito comuns em grande parte das aplicações web e portanto é importante saber.

Capítulo 12

Polindo o Colcho.net

Não importa se você vai devagar, o que importa é você nunca parar
– Confúcio

Este é o último capítulo em que vamos desenvolver funcionalidades no Colcho.net. Você está chegando ao final, e a aplicação já está bem funcional. Nós vamos fazer algumas melhorias na aplicação, desenvolvendo as seguintes funcionalidades: busca textual, *URL slugs* usando a gem `friendly_id`, paginação usando a *gem* `kaminari`, *upload* de fotos usando a *gem* `carrierwave` e, finalmente, colocar o Colcho.net online, no Heroku.

12.1 Faça buscas textuais apenas com o Rails

No Colcho.net há três informações importantes pelas quais um usuário pode pesquisar: localidade, título e a descrição de um quarto. Precisamos de uma busca que nos permita pesquisar nesses três "textos", ou seja, precisamos de uma busca textual, um pouco diferente das buscas que temos feito até então.

Busca textual é um assunto **bastante** complexo e existe muita literatura sobre o assunto, portanto vamos fazer uma funcionalidade simplificada e cobrir o que é possível criar com apenas o uso do Rails.

A funcionalidade é a seguinte: vamos criar um campo de texto próximo ao título de listagem de quartos. Buscaremos também o conteúdo da caixa de texto e procurar nos campos mencionados anteriormente. Dos resultados encontrados, vamos marcar visualmente para o usuário facilmente identificar o trecho que está procurando.

Primeiro, vamos construir a funcionalidade no *back-end*. Para isso, vamos criar um método chamado `.search` no modelo de quartos (`app/models/room.rb`):

```
class Room < ActiveRecord::Base
  # ...
  def self.search(query)
    if query.present?
      where(['location LIKE :query OR
              title LIKE :query OR
              description LIKE :query', query: "%#{query}%"])
    else
      scoped
    end
  end
  # ..
end
```

Neste método, se a busca estiver presente, fazemos o filtro usando o operador `LIKE` do SQL para fazer a busca nos campos mencionados: `location`, `title` e `description`. Caso contrário, vamos retornar o escopo atual:

```
Room.search('Sao')
# SELECT "rooms".* FROM "rooms" WHERE (location LIKE '%Sao%' OR
#  title LIKE '%Sao%' OR
#  description LIKE '%Sao%')
# => <ActiveRecord::Relation [#<Room id: 4, ... >]>

Room.search('')
# SELECT "rooms".* FROM "rooms"
# => <ActiveRecord::Relation [#<Room id: 4, ... >]>
```

Por que usar scoped e não all?

Seria natural pensar em usar o método `.all` no caso da busca não estiver presente, pois retorna todos os objetos. Porém, se este método for usado em conjunto com outros escopos, o que é bastante comum, o `.all` iria cancelar todos os outros escopos e o comportamento final seria bem inesperado. O `.scoped` faz esse papel de manter o escopo.

Como o modelo já faz todo o trabalho pesado, no controle de quartos (`app/controllers/rooms_controller.rb`) vamos apenas chamar o método que acabamos de criar. A ação `index` fica da seguinte forma:

```
class RoomsController < ApplicationController
  # ...

  def index
    @search_query = params[:q]

    rooms = Room.search(@search_query)
    @rooms = rooms.most_recent.map do |room|
      RoomPresenter.new(room, self, false)
    end
  end

  # ...
end
```

Vamos diferenciar o *template* de índice (`app/views/rooms/index.html.erb`) apenas para mostrar um título diferente caso estejamos exibindo resultado de buscas. Para isso, vamos verificar a presença de um termo de busca (`search_query`). Vamos incluir no topo do *template* o formulário de busca. O resultado é:

```
<%= render 'search' %>

<h1>
  <% if @search_query.present? %>
    <%= t '.search_results' %>
  <% else %>
```

```
    <%= t '.title' %>
  <% end %>
</h1>

<%= render @rooms %>
```

Na *partial* contendo o formulário (app/views/rooms/_search.html.erb) não será possível usar o *helper* form_for, pois não estamos criando um formulário de modelo. O Rails possui também *helpers* para esta situação:

```
<%= form_tag rooms_path, method: :get, class: 'search' do %>
  <%= text_field_tag :q, @search_query,
        placeholder: t('.search_for') %>
<% end %>
```

O que fazemos é enviar os dados à ação index do controle de quartos via o parâmetro q, que possui o conteúdo para filtrar a listagem. Observe que temos que declarar o método HTTP para GET para que o roteador não nos envie para a ação create.

Atualizamos as chaves I18n (config/locales/pt-BR.yml), para adicionar as mensagens relativas às buscas:

```
rooms:
  index:
    title: 'Quartos disponíveis'
    search_results: 'Resultados da busca'
  search:
    search_for: 'Buscar por...'
```

Em seguida, usamos o *helper* highlight do Rails, para destacar os resultados da busca, na *partial* de quarto. Alteraremos o título, a descrição e a localidade do quarto (app/views/rooms/_room.html.erb):

```
<article>
...
  <h2>
    <%= link_to highlight(room.title, @search_query), room.route %>
  </h2>
...
  <p>
    <span class="location">
```

```
      <%= link_to highlight(room.location, @search_query),
                  "https://maps.google.com/?q=#{room.location}",
                  target: :blank %>
    </span>
  </p>
  <p><%= highlight(room.description || '', @search_query) %></p>
...
</article>
```

Por fim, adicionamos o estilo CSS para o campo de busca. Usamos a "Font Awesome" para adicionar o clássico ícone da lupa (`app/assets/stylesheets/room.css.scss`), e vamos estilizar os campos destacados pela função `highlight`:

```
#content .search {
  float: right;
  margin: 0;
  position: relative;
  &:after {
    padding: 5px;
    font-size: 14px;
    font-family: 'FontAwesome';
    position: absolute;
    content: "\f002";
    color: #bbb;
    top: 0;
    right: 0;
  }
}

.highlight {
  font-size: inherit;
  font-weight: inherit;
  background-color: gold;
}
```

Com essas alterações, ao realizarmos uma pesquisa, seu resultado é exibido com o destaque no termo procurado.

Figura 12.1: Resultado de buscas destacados

Por fim, é necessário lembrar que essa busca é uma busca simples. Se você quiser fazer buscas mais rebuscadas, é interessante usar tecnologias como o Solr (http://lucene.apache.org/solr/) . Com o Solr, você é capaz de criar índices complexos, fazer busca facetada, filtros e outras funcionalidades avançadas. Se você tiver interesse, verifique a *gem* `sunspot` (https://github.com/outoftime/sunspot) , que integra o Rails com o Solr.

12.2 URLs mais amigáveis através de slugs

Quando você acessa o site da Casa do Código, para ver esse livro, você entra no endereço http://www.casadocodigo.com.br/products/livro-ruby-on-rails . Internamente, esse livro possui um ID, e poderia ser usado para busca. Acontece que essa URL não é nada amigável, números soltos dessa forma não traz nenhum significado ao usuário. Chamamos essa técnica de mapear nomes amigáveis à IDs de *URL Slugs*

URL slugs são URLs que não usam IDs e sim uma URL mais interessante. Ou seja, ao invés de usar algo como http://localhost:3000/rooms/4 , usamos http://localhost:3000/rooms/colchonete-na-sala . Esta técnica também traz vantagens na otimização para mecanismos de busca (ou *SEO - Search Engine Optimization*).

Para implementar essa funcionalidade, podemos alterar a forma que as *URL helpers* do Rails constroem as rotas a partir do modelo. Os *helpers* chamam o método `#to_param` de modelos. Esse método é usado para criar uma string que representa o objeto em URLs. A implementação padrão do Rails irá retornar o ID do objeto:

```
room = Room.first
# => #<Room id: 4, ... >

room.to_param
# => "4"
```

Além disso, podemos aproveitar o fato de que, em Ruby, strings são ignoradas em conversões de números:

```
"123-isso-vai-desaparecer".to_i
# => 123
```

Isso significa que, quando uma busca for feita por ID, apenas a parte numérica é utilizada.

Portanto, podemos apenas então alterar o método `to_param` no modelo de quartos para incluir uma descrição amigável do quarto, e tudo continuará funcionando:

```
class Room < ActiveRecord::Base
  # ...

  def to_param
    "#{id}-#{title.parameterize}"
  end
end

Room.find(2)
# => #<Room id: 2, title: "Casa de praia completa", ...>

Room.find('2-casa-de-praia-completa')
# => #<Room id: 2, title: "Casa de praia completa", ...>
```

Nesse exemplo, o link gerado é: http://localhost:3000/rooms/2-casa-de-praia-completa

Essa solução funciona, porém, há vários detalhes que temos que tomar cuidado com essa implementação. Por exemplo, o que fazer com conflitos de *slugs*? Mais ainda, como lidar com detalhes difíceis de prever, tal como transliteração de caracteres, ou seja, transformar símbolos como "ø" ou "ã" em "oe" e "a"?

Portanto, ao invés de ter que nos preocupar com estas situações (e outras não listadas aqui também), podemos nos aproveitar da experiência de outros desenvolvedores, usando ferramentas `open source`.

Para resolver o problema de *slugs*, vamos usar a *gem* `friendly_id`, feita pelo Norman Clarke, desenvolvedor que atua na comunidade Ruby e Rails há anos. A ferramenta encontra-se em: https://github.com/FriendlyId/friendly_id

Para instalar esta *gem*, o primeiro passo é adicioná-la no `Gemfile`:

Cuidado com gems em desenvolvimento!

Infelizmente a gem ainda não está completamente preparada para uso com o Rails 4. Vamos usá-la de qualquer forma pois é uma questão de pouco tempo para que ela se torne estável e sua versão 5 seja liberada.

Se for utilizar a gem `friendly_id`, lembre-se de remover o método `to_param` que implementamos manualmente.

```
gem 'friendly_id', github: 'FriendlyId/friendly_id', branch: 'master'
```

E usar o bundler para instalar as dependências:

```
$ bundle
...
Fetching git://github.com/FriendlyId/friendly_id.git
...
Using friendly_id (5.0.0.alpha1)
   from git://github.com/FriendlyId/friendly_id.git (at master)
...
```

Em seguida, a *gem* precisa que você crie um campo extra, chamado `slug`, na tabela quarto, que é a informação que teremos a URL amigável. Para isso, vamos criar uma nova migração:

```
$ rails g migration add_slugs_to_rooms slug:string:index
 invoke  active_record
 create    db/migrate/20130719090033_add_slugs_to_rooms.rb
```

Precisamos adicionar um índice de unicidade. Por isso, alteraremos a migração gerada:

```
class AddSlugsToRooms < ActiveRecord::Migration
  def change
    add_column :rooms, :slug, :string
```

```
    add_index :rooms, :slug, unique: true
  end
end
```

Vamos também ter que criar uma tabela para guardar os *slugs* antigos. O `friendly_id` já faz isso para nós, basta executar:

```
$ rails generate friendly_id
db/migrate/20130719093530_create_friendly_id_slugs.rb
```

Executamos as migrações:

```
$ rake db:migrate
```

Alteraremos o modelo quarto (`app/models/room.rb`) para usar o `friendly_id`. Para isso, segundo a documentação da gem, é necessário estender o módulo `FriendlyId`, adicionar uma validação de modo que *slugs* sejam obrigatórios no banco de dados e por fim, configurar o módulo `friendly_id`, para usar as funcionalidades `:slugged`, que é o modo padrão de operação da *gem* e a `:history`, para gravar o histórico de *slugs*:

```
class Room < ActiveRecord::Base
  extend FriendlyId

  #...
  validates_presence_of :title
  validates_presence_of :slug

  friendly_id :title, use: [:slugged, :history]

  #...
end
```

Por fim, precisamos alterar a busca de quartos. Para isso, mudaremos todas as chamadas de `Room.find` para `Room.friendly.find` no controle de quartos:

```
class RoomsController < ApplicationController
  before_action :require_authentication,
    only: [:new, :edit, :create, :update, :destroy]

  def index
    @search_query = params[:q]
```

```
    rooms = Room.search(@search_query)
    @rooms = rooms.most_recent.map do |room|
      # Não exibiremos o formulário na listagem
      RoomPresenter.new(room, self, false)
    end
  end

  def show
    room_model = Room.friendly.find(params[:id])
    @room = RoomPresenter.new(room_model, self)
  end

  def new
    @room = current_user.rooms.build
  end

  def edit
    @room = current_user.rooms.friendly.find(params[:id])
  end

  def create
    @room = current_user.rooms.build(room_params)

    if @room.save
      redirect_to @room, notice: t('flash.notice.room_created')
    else
      render action: "new"
    end
  end

  def update
    @room = current_user.rooms.friendly.find(params[:id])

    if @room.update(room_params)
      redirect_to @room, notice: t('flash.notice.room_updated')
    else
      render action: "edit"
    end
  end
```

```
  def destroy
    @room = current_user.rooms.friendly.find(params[:id])
    @room.destroy

    redirect_to rooms_url
  end
  private

  def room_params
    params.
      require(:room).
      permit(:title, :location, :descriptions)
  end
end
```

Pronto! Você já tem a funcionalidade de *URL slugs*. Para testar, crie um novo quarto e veja a URL como fica (lembre-se de reiniciar o servidor do Rails, caso já não tenha feito).

Atualizando quartos já cadastrados

O `FriendlyId` irá funcionar com quartos sem *slug*, ou seja, irá usar o ID do modelo caso não seja possível usar o *slug*. Porém, para termos consistência, podemos atualizar os modelos para usarem *slugs*, basta ativar os *callbacks* de `save` que o `FriendlyId` irá fazer o resto:

```
Room.find_each(&:save)
```

O `find_each` é uma alternativa ao `all` para fazer alterações em lotes, caso você tenha muitos registros no banco de dados.

12.3 Paginação de dados de forma descomplicada

Imagine o Colcho.net no futuro, explodindo de sucesso e com mais de 500 quartos cadastrados. A listagem ficará lenta, pois são muitos objetos para se exibir e calcular a média da pontuação. Para essa situação, é normal paginar o resultado das buscas, de modo que os resultados menos relevantes fiquem ao final.

Essa funcionalidade não é complicada de se criar usando escopos. Mas a *gem* `kaminari` (https://github.com/amatsuda/kaminari) torna isso ainda mais fácil.

Vamos primeiro instalar a *gem*. Como de costume, basta colocá-la no `Gemfile` e fazer o `bundle`:

```
gem 'kaminari'
```

```
$ bundle
...
Installing kaminari (0.14.1)
...
```

O `kaminari` insere, através de meta-programação, alguns métodos no `ActiveRecord` para fazer a paginação. Porém, o uso de *presenters* irá atrapalhar, já que estamos usando esses objetos ao invés dos modelos diretamente.

Precisamos criar um outro *presenter* para que possamos chamar os métodos de paginação do `kaminari`. Outra coisa que precisamos fazer é tornar essa nova classe uma espécie de coleção, para que o `render collection: @rooms` continue funcionando.

Infelizmente o Rails não definiu uma interface para este tipo de interação. Portanto precisamos implementar o método `#to_ary`, que faz uma conversão do objeto atual para Array quando isso for necessário (via conversão implícita). Veja o exemplo a seguir:

```
class ImplicitArray
  def to_ary
    [1,2,3]
  end
end
# => nil
[:a] + ImplicitArray.new
# => [:a, 1, 2, 3]
```

Repare que ao juntarmos o Array de `:a`, com a instância de `ImplicitArray`, o resultado foi um novo Array com a junção de `:a` e o resultado do método `to_ary` de `ImplicitArray`, chamado pelo próprio Ruby.

Com isso em mente, vamos criar o *presenter* `RoomCollectionPresenter` (`app/presenters/room_collection_presenter.rb`), que irá delegar os métodos de paginação para a coleção `ActiveRecord` e criar *presenters* de quarto quando o *collection presenter* for convertido para Array.

```
class RoomCollectionPresenter
  delegate :current_page, :num_pages, :limit_value, :total_pages,
    to: :@rooms

  def initialize(rooms, context)
    @rooms = rooms
    @context = context
  end

  def to_ary
    @rooms.map do |room|
      RoomPresenter.new(room, @context, false)
    end
  end
end
```

Reuso dos presenters

Este é o tipo de código que sempre precisamos, portanto você pode aplicar em seus próprios projetos. Se preferir usar um *presenter* mais completo, vale a pena checar a *gem* `draper`: https://github.com/drapergem/draper

Alteramos a ação `index` do `RoomsController` (`app/controllers/rooms_controller.rb`). Usaremos a paginação no modelo de quarto, via o uso dos métodos `.page`, que define a página a ser buscada e o método `.per`, que define a quantidade de objetos por página. Na última linha da ação, embrulhamos a coleção no *presenter* que acabamos de criar:

```
class RoomsController < ApplicationController
  PER_PAGE = 10

  # ...

  def index
    @search_query = params[:q]
    rooms = Room.search(@search_query).
      page(params[:page]).
      per(PER_PAGE)
```

```
    @rooms = RoomCollectionPresenter.new(rooms.most_recent, self)
  end
end
```

Com isso pronto, para usar a paginação do `kaminari`, basta colocar o *helper* `paginate` no *template* da ação `index` (`app/views/rooms/index.html.erb`):

```
...
<%= render @rooms %>

<%= paginate @rooms %>
```

Isso é tudo que precisamos para fazer a paginação funcionar.

Vamos melhorar um pouco mais, criando um novo conjunto de chaves I18n para o `kaminari`, no arquivo `config/locales/pt-BR.pagination.yml`:

```
pt-BR:
  views:
    pagination:
      first: "&laquo; Primeira"
      last: "Última &raquo;"
      previous: "&lsaquo; Anterior"
      next: "Próxima &rsaquo;"
      truncate: "..."
```

Por fim, um pouco de CSS para ajustar a exibição da paginação (`app/assets/stylesheets/default.css.scss`):

```
.pagination {
  margin: 20px 0;
}
.pagination span {
  padding: 5px;
  margin: 0 3px;
  background-color: #fff;
  border: 1px solid #ccc;
  @include shadow(#ccc, 0, 3px, 6px);
}
```

O resultado é o seguinte:

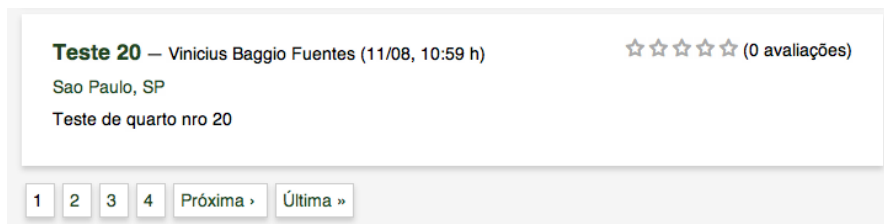



Figura 12.2: Links para paginação no fim da página

Pronto, agora você tem uma paginação feita, sem trazer dados demais para a memória e sem exibir informações demais na tela.

12.4 Upload de fotos de forma simples

Imagine você alugando um quarto ou um apartamento sem conseguir ver fotos desse lugar? Difícil, não? Para isso, precisamos permitir que fotos sejam associadas ao quarto, através de upload de arquivos de imagens.

Para fazer o *upload* de fotos de quartos, vamos usar uma *gem* chamada `carrierwave`. Com ela é possível criar *thumbnails* de fotos automaticamente e até enviar essa foto para ser servida de serviços como o S3, da Amazon (http://aws.amazon.com/s3/) ou Cloud Files, da Rackspace (http://www.rackspace.com/cloud/public/files/) .

Para instalá-la, a receita é parecida: colocar a *gem* no `Gemfile` e fazer o `bundle`. A diferença é que temos que declarar a dependência à gem `rmagick` manualmente. Isso deve-se ao fato de que o `carrierwave` funciona com outras *gems* de processamento de imagem.

```
gem 'carrierwave'
gem 'rmagick'
```

```
$ bundle
...
Installing carrierwave (0.8.0)
Installing rmagick (2.13.2) with native extensions
...
```

O funcionamento do `carrierwave` é o seguinte: primeiro, é necessário criar um *uploader*, uma classe com a descrição das transformações que a imagem vai passar (criação de *thumbnails*, por exemplo) e onde guardar o arquivo enviado pelo

usuário.. Para facilitar esse trabalho, o `carrierwave` instala um gerador. Então, vamos criar o *uploader* para fotos de quartos:

```
$ rails g uploader Picture
 create   app/uploaders/picture_uploader.rb
```

O arquivo gerado possui diversos comentários sobre a forma de utilizar o *uploader* e os parâmetros configuráveis. O resultado, das linhas não comentadas deve ser o seguinte:

```
# encoding: utf-8

class PictureUploader < CarrierWave::Uploader::Base
  include CarrierWave::RMagick
  include Sprockets::Helpers::RailsHelper

  storage :file

  # Diretório onde os arquivos serão armazenados
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Redimensiona a imagem para ficar no tamanho de
  # no máximo 500x500, mantendo o aspecto e cortando
  # a imagem, se necessário.
  process :resize_to_fill => [500, 500]

  # Dimensões do thumbnail
  version :thumb do
    process :resize_to_fill => [100, 100]
  end

  # Informa os formatos permitidos
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

Carrierwave não é só para imagens

Apesar de conter muitas facilidades para o *upload* de imagens, ele pode ser usado com qualquer outro formato de arquivo, desde que você desative as funcionalidades específicas para imagens, como processamento de tamanho. As outras ferramentas de upload para serviços ainda funcionam.

Uma vez criado o *uploader*, precisamos criar uma coluna no banco de dados para que o `carrierwave` guarde o nome do arquivo e saiba recuperá-lo na hora de exibir a foto. Para isso, criemos e executemos a migração a seguir:

```
$ rails g migration add_picture_to_rooms picture
  create    db/migrate/20120813014045_add_picture_to_rooms.rb

$ rake db:migrate
==  AddPictureToRooms: migrating ==============
-- add_column(:rooms, :picture, :string)
   -> 0.0012s
==  AddPictureToRooms: migrated (0.0013s) =====
```

Em seguida, precisamos associar o *uploader* ao modelo quarto. Isso é feito através da *class macro* `mount`, método que o `carrierwave` adiciona ao `ActiveRecord`. Portanto, no modelo quarto (`app/models/room.rb`), basta adicionar o seguinte código:

```
class Room < ActiveRecord::Base
  # ...

  mount_uploader :picture, PictureUploader
  friendly_id :title, use: [:slugged, :history]

  # ...
end
```

Essa *class macro* irá tornar o campo `picture` do modelo quarto em um `PictureUploader`, ao invés de uma simples `string`. A partir daí, basta colocarmos mais um campo no formulário de quartos, para que o arquivo seja informado. Então, no `app/views/rooms/_form.html.erb`:

```
...
<%= form_for(@room) do |f| %>
  ...
  <p>
    <%= f.label :picture %>
    <%= f.file_field :picture %>
    <%= error_tag @room, :picture %>
  </p>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Adicione também a chave para traduzir o novo campo no arquivo de I18n (`config/locales/pt-BR.yml`):

```
pt-BR:
  #...
  activerecord:
    #...
    attributes:
      # ...
      room:
        description: Descrição
        location: Localização
        title: Título
        picture: Foto
```

Para que as fotos possam ser gravadas, precisamos adicionar o campo `picture` na lista de atributos permitidos pelo controle. Altere o método `room_params`, na classe `RoomsController`:

```
def room_params
  params.
    require(:room).
    permit(:title, :location, :descriptions, :picture)
end
```

Por fim, vamos alterar o *presenter*, HTML e o CSS para exibir a imagem. Altere o *presenter* de quartos para incluir os métodos que verificam se há uma imagem, preparam o *thumbnail* e a própria imagem (`app/presenters/room_presenter.rb`):

```
class RoomPresenter
  # ...
  def picture_url
    @room.picture_url
  end

  def thumb_url
    @room.picture.thumb.url
  end

  def has_picture?
    @room.picture?
  end
end
```

O *template* de quartos (app/views/rooms/_room.html.erb) fica assim:

```
<article class="room">
  ...

  <div class="room-description">
    <%= link_to(
          image_tag(room.thumb_url),
          room.picture_url
        ) if room.has_picture? %>

    <p><%= highlight(room.description || '', @search_query) %></p>
  </div>
  ...
</article>
```

O CSS para a imagem de quartos (app/assets/stylesheets/room.css.scss) é:

```
.room-description {
  p, img {
    float: left;
    margin-right: 10px;
  }
}
.room-description + ul {
  padding-top: 10px;
```

```
  clear: both;
}
```

Et voilà! Os quartos agora possuem foto:

Figura 12.3: Quarto com uma foto

12.5 Coloque a aplicação no ar com o Heroku

Antigamente, colocar um aplicativo web no ar era complicado e exigia bastante conhecimento de configuração de servidor. Hoje em dia, contudo, existem serviços que oferecem hospedagem de aplicativos sem a noção de termos um servidor. *Platform as a service - PaaS*, ou "Plataforma como serviço", é uma das facetas de *Cloud Computing*, na qual um usuário contrata uma plataforma e poder de processamento, ao invés de contratar servidores, sejam de metal ou virtual.

Um desses serviços é o Heroku (www.heroku.com). Não existe forma mais fácil de colocar uma aplicação Rails no ar do que usar o Heroku. Por isso, vamos usá-lo: é de graça para pouco processamento de requisição (1 requisição por segundo), que é o bastante para nossa aplicação. Antes de continuar, crie sua conta no site, é bastante simples.

Preparando a aplicação para o Heroku

É necessário fazer algumas alterações no aplicativo para tornar compatível com o serviço: o Heroku funciona com PostgreSQL e estamos usando o SQLite3. Por esse

motivo, é necessário colocar a *gem* `pg` no `Gemfile`, responsável pela conectividade a bancos PostgreSQL e alterar uma consulta SQL. Primeiro, vamos a alteração do `Gemfile` e execute `bundle` em seguida:

```
# gem 'sqlite3'
gem 'pg'
```

```
$ bundle
...
Installing pg (0.14.1) with native extensions
...
```

Vamos agora alterar a consulta SQL problemática: no PostgreSQL, a consulta `LIKE` é sensível a maiúsculas e minúsculas, enquanto a `ILIKE` não. Vamos alterar o método `Room.search` (`app/models/room.rb`) para usar a nova consulta:

```
class Room < ActiveRecord::Base
  # ...

  def self.search(query)
    if query.present?
      where(['location ILIKE :query OR
              title ILIKE :query OR
              description ILIKE :query', query: "%#{query}%"])
    else
      scoped
    end
  end

  # ...
end
```

Desenvolvimento usando PostgreSQL

Infelizmente esse comportamento do `LIKE` vs. `ILIKE` é incompatível no SQLite. Por isso, pode ser interessante desenvolver diretamente no PostgreSQL.

Se você usa Linux, verifique os pacotes de sua distribuição. De preferência, instale uma versão igual ou superior a 9. No OS X, para usuários de 10.7 (Lion) ou superior, você já tem o PostgreSQL. Para Windows, você pode usar o instalador do site oficial: http://postgresql.org.

Em seguida, altere a parte `development` e `test` do arquivo `config/database.yml` para refletir as alterações. Não é necessário ter `production`, ele será criado pelo próprio Heroku. Veja o exemplo de como deve ficar o arquivo, lembrando de trocar `vinibaggio` pelo seu usuário de sistema:

```
development:
  adapter: postgresql
  database: colchonet_dev
  host: localhost
  username: vinibaggio
  password:
  pool: 5
  timeout: 5000


test:
  adapter: postgresql
  database: colchonet_test
  host: localhost
  username: vinibaggio
  password:
  pool: 5
  timeout: 5000
```

Por fim, execute:

```
$ rake db:create db:migrate
==  CreateRooms: migrating ====================
-- create_table(:rooms)
...
==  AddPictureToRooms: migrating ==============
-- add_column(:rooms, :picture, :string)
   -> 0.0011s
==  AddPictureToRooms: migrated (0.0012s) ======
```

Colocando a aplicação no ar

Para começar a usar o Heroku, é necessário instalar o `git`, criar uma conta no Heroku e baixar o Heroku Tools. Uma vez com tudo instalado, basta ir na pasta do projeto, e criar um repositório `git` e fazer um *commit*:

```
$ git init .
Initialized empty Git repository in /book/code/colchonet/.git/
```

```
$ git add .

$ git commit -m "Primeiro commit"
[master (root-commit) a2ed4be] Primeiro commit
 106 files changed, 3169 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Gemfile
 create mode 100644 Gemfile.lock
 create mode 100644 README.rdoc
 ...
```

O que é git? Como instalar?

O `git` é um sistema de versionamento de código-fonte. Se você já ouviu falar em CVS, SVN ou Mercurial, o `git` é uma ferramenta similar. Para saber mais, veja o *screencast* "Começando com Git", do Fábio Akita, em http://colcho.net/comecando-com-git. Nele você poderá saber como funciona o `git` e como instalá-lo.

Uma vez com o *commit* criado, vamos usar o `heroku tools` para criar um repositório remoto chamado `heroku` e preparar o nosso novo site.

```
$ heroku login
Enter your Heroku credentials.
Email: XXX@XXX.com
Password (typing will be hidden):
Authentication successful.

$ heroku create
Creating calm-badlands-8648... done, stack is cedar
http://calm-badlands-8648.herokuapp.com/ |
             git@heroku.com:calm-badlands-8648.git
Git remote heroku added
```

Precisamos adicionar um *add-on* ao novo site para que possamos entregar os e-mails de cadastro. Vamos usar o Mailgun, é de graça e possui integração simples com o Heroku:

```
$ heroku addons:add mailgun:starter
Adding mailgun:starter on calm-badlands-8648... done, v8 (free)
Use `heroku addons:docs mailgun:starter` to view documentation.
```

Precisamos atualizar o ambiente `production` (`config/environments/production.rb`), pois é neste ambiente que o Heroku executa a nossa aplicação. Precisamos atualizar as configurações do `ActionMailer` para usar a nova URL do site e as configurações do Mailgun (conforme vimos na seção 8.2):

```
Colchonet::Application.configure do
  #...

  config.action_mailer.default_url_options = {
    host: "fierce-everglades-2744.herokuapp.com"
  }

  config.action_mailer.delivery_method = :smtp
  config.action_mailer.smtp_settings = {
    port:           ENV['MAILGUN_SMTP_PORT'],
    address:        ENV['MAILGUN_SMTP_SERVER'],
    user_name:      ENV['MAILGUN_SMTP_LOGIN'],
    password:       ENV['MAILGUN_SMTP_PASSWORD'],
    domain:         'fierce-everglades-2744.herokuapp.com',
    authentication: :plain,
  }
end
```

As configurações do Mailgun são colocadas como variáveis de ambiente, bastante conveniente para não termos que gerenciar chaves de API. Em seguida, execute o seguinte comando, que irá publicar a aplicação:

```
$ git push heroku master
Counting objects: 172, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (135/135), done.
Writing objects: 100% (172/172), 988.80 KiB | 267 KiB/s, done.
Total 172 (delta 22), reused 172 (delta 22)

-----> Ruby/Rails app detected
-----> Using Ruby version: ruby-2.0.0
```

```
...
-----> Discovering process types
       Procfile declares types      -> (none)
       Default types for Ruby/Rails -> console, rake, web, worker
-----> Compiled slug size is 12.4MB
-----> Launching... done, v4
       http://calm-badlands-8648.herokuapp.com deployed to Heroku

To git@heroku.com:calm-badlands-8648.git
 * [new branch]      master -> master
```

Quase pronto... Por fim, temos que migrar o banco de dados recém criado:

```
$ heroku run rake db:migrate
Running `rake db:migrate` attached to terminal... up, run.1
Connecting to database specified by DATABASE_URL
Migrating to CreateRooms (20120610045608)
...
```

Depois que o comando terminar, basta acessar a URL que você recebeu e já pode passar para sua família e amigos! Muito bom, não é mesmo? Parabéns! Você completou o Colcho.net!

Capítulo 13

Próximos passos

Agora, isso não é o fim. Nem sequer é o início do fim. Mas é, talvez, o fim do começo.
– Sir Winston Churchill

Agora que você já é familiar ao Rails, está na hora de fazer as suas próprias aplicações. Seria muita pretensão deste livro te ensinar tudo que existe no ecossistema Rails, portanto, depois da leitura deste livro, ainda é necessário buscar mais material.

Lista de email

Se você quer tirar alguma dúvida sobre este livro, você pode se juntar à lista de emails do livro em http://colcho.net/lista e poderá enviar sua pergunta. Os participantes dela, incluindo eu, o autor, tentarão te ajudar. Lembre-se sempre de ser educado.

Conhecimentos de Rails

O primeiro lugar que você deve buscar para tirar dúvidas e entender o fun-

cionamento específico de algum componente do Rails são os RailsGuides (http://guides.rubyonrails.org) . Nele você poderá buscar informações sobre os principais componentes do Rails, tais como `ActiveRecord`, *templates*, etc.

Se sua dúvida for mais específica, talvez seja interessante consultar a documentação da API, que fica em http://api.rubyonrails.org. A documentação é fácil de navegar, bastando que você saiba o método que quer procurar.

Se você quiser aprender receitas de como resolver certos problemas, o Ryan Bates, bastante famoso na comunidade Rails, faz o RailsCasts (http://railscasts.com) . Cada *screencast* é uma receita de como resolver um problema, e o inglês que ele fala é claro, sendo uma ótima maneira de aprender. Se você ainda não está confortável em entender uma narração em inglês, pode ler os AsciiCasts (http://asciicasts.com/) , que é praticamente uma transcrição do RailsCasts.

Se você prefere livros, terá que aguardar um pouco. A maioria dos livros para o Rails 4 ainda está em Beta, ainda não finalizados, e todos em inglês. Se você estiver confortável em comprar um livro ainda em beta, recomendo o "Crafting Rails 4 Applications" (http://colcho.net/crafting-rails-apps) . Neste livro, o José Valim, um dos principais desenvolvedores do próprio *framework*, te guia para um mergulho de cabeça nas profundidades do Rails. Pode ser um pouco difícil para iniciantes, mas é um livro de altíssima qualidade e te dará conhecimento o suficiente para conseguir fazer funcionalidades bem complexas.

Dominando o Ruby

Se você quer se tornar um desenvolvedor Ruby e Rails profissional, recomendo aprofundar-se na linguagem, pois é de extrema importante um profissional conhecer bem as ferramentas que está trabalhando. Minha recomendação para este fim é o livro de Ruby, da Casa do Código: http://www.casadocodigo.com.br/products/livro-ruby, totalmente em português.

Outro livro que pode ser interessante é o "Eloquent Ruby", do Russ Olsen (http://colcho.net/eloquent-ruby)

Dominando testes

A comunidade Ruby e Rails em geral gosta bastante de testes unitários e argumentam que esta prática melhora o *design* de código e ajuda a reduzir o número de *bugs*. É interessante entender as ideias, por isso recomendo a leitura do "Test-Driven Development", do Kent Beck (http://colcho.net/tdd) . Em se-

guida, é interessante ler como TDD se aplica no Ruby, com RSpec e Cucumber. Em português, o meu amigo Hugo Baraúna está trabalhando no Cucumber e Rspec, publicado pela Casa do Código http://www.casadocodigo.com.br/products/livro-cucumber-rspec-tdd-bdd. Outro livro interessante, em inglês, é o "The RSpec Book" (http://colcho.net/rspec-book) .

É difícil no começo, não se preocupe se tiver dificuldades, é normal. Um pouco de perseverança e você poderá sentir os benefícios dessa prática. Fique à vontade de mandar perguntas sobre esse assunto na lista de discussões desse livro.

Envolvimento na comunidade

A comunidade rubista brasileira é bastante ativa. Em São Paulo, por exemplo, existe o GURU-SP, porém é possível encontrar outros grupos de Ruby e Rails pelo Brasil e pelo mundo. Procure as listas de email, participe e fique atento aos encontros, você pode aprender muito! Não deixe de participar também em outros eventos de programação que existem no Brasil, pois exposição a tecnologias e novas ideias sempre te ajudarão com qualquer linguagem ou *framework*.

Open source e leitura de código

A comunidade de Ruby e Rails é bastante envolvida em *open source* e se envolver em um projeto é uma grande forma de, além de contribuir, aprender com outros desenvolvedores. A leitura de código também é encorajada. Uma ótima maneira é procurar sua *gem* favorita no GitHub (http://www.github.com) e navegar pelo código fonte.

Índice Remissivo

Referências Bibliográficas

[1] David Thomas Andrew Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[2] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.

[3] Zach Dennis Aslak Hellesøy Bryan Helmkamp Dan North David Chelimsky, Dave Astels. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2010.

[4] Ian Robinson Jim Webber, Savas Parastatidis. *Rest in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.

[5] Nando Vieira. Guia rápido de rspec. http://colcho.net/guia-rspec, 2010.